

Advanced consolidation for dynamic containers

Damien Carver

▶ **To cite this version:**

Damien Carver. Advanced consolidation for dynamic containers. Operating Systems [cs.OS]. Sorbonne Université, 2019. English. NNT : 2019SORUS513 . tel-02393773v3

HAL Id: tel-02393773
https://theses.hal.science/tel-02393773v3

Submitted on 12 Feb 2021

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.





ÉCOLE DOCTORALE EDITE DE PARIS (ED130)
INFORMATIQUE, TÉLÉCOMMUNICATION ET ÉLECTRONIQUE

THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

SPÉCIALITÉ : **INGÉNIERIE / SYSTÈMES INFORMATIQUES**

PRÉSENTÉE PAR : **DAMIEN CARVER**

POUR OBTENIR LE GRADE DE :

DOCTEUR DE SORBONNE UNIVERSITÉ

SUJET DE LA THÈSE :

ADVANCED CONSOLIDATION FOR DYNAMIC CONTAINERS

SOUTENUE LE 17 MAI 2019

DEVANT LE JURY COMPOSÉ DE :

Rapporteurs :	LAURENT RÉVEILLÈRE	Professeur, Université de Bordeaux
	GILLES GRIMAUD	Professeur, Université de Lille 1
Examinateurs :	GILLES MULLER	DR Inria
	ALAIN TCHANA	Professeur, Université de Nice
	BÉATRICE BÉRARD	Professeur, Sorbonne Université
	JEAN-PIERRE LOZI	Principal Member of the Technical Staff, Oracle Labs
Encadrants :	SÉBASTIEN MONNET	Professeur, Université Savoie Mont Blanc
	JULIEN SOPENA	Maître de conférences, Sorbonne Université
	DIMITRI REFAUVELET	C.T.O., Magency





Thank you so much,
It would have been impossible without you.

CONTENTS

Contents **3**

1 Introduction **7**

2 Resource Virtualization **11**
2.1 Virtual Machines: Hardware-level virtualization 11
2.1.1 Cost of machine virtualization 12
2.1.2 Improving utilization in a full virtualization context 13
2.1.3 Improving utilization in a paravirtualization context 13
2.2 Containers: Operating System-level virtualization 13
2.2.1 Isolating physical resources with cgroups 14
2.2.2 Isolating resource visibility with namespaces 17
2.2.3 Restraining attack surface with security features 20
2.3 Containers and VMs comparison 22
2.3.1 Comparing stand-alone overheads 23
2.3.2 Comparing performance isolation and overcommitment 23
2.3.3 Should VMs mimic Containers? 24
2.4 Consolidation and Isolation, the best of both worlds 25
2.4.1 Resource Consolidation 26
2.4.2 Performance Isolation 26
2.4.3 Illustrating Consolidation and Isolation with the CPU cgroups 26
2.4.4 Block I/O, a time-based resource similar to CPU 29
2.5 Memory, a spatial but not time-based resource 31
2.5.1 Conclusion 33

3 Memory and cgroup **35**
3.1 Storing data in main memory 35
3.1.1 Memory Hierarchy 35
3.1.2 Spatial multiplexing 36
3.1.3 Temporal multiplexing 36
3.1.4 The need for memory cgroup 37
3.2 Accounting and limiting memory with cgroup 37
3.2.1 Event, Stat and Page counters 38
3.2.2 `min`, `max`, `soft` and `hard` limits 40

3.3 Isolating cgroup memory reclaims . . . 41
3.3.1 Linux memory pool . . . 41
3.3.2 Splitting memory pools . . . 42
3.4 Resizing dynamic memory pools . . . 44
3.4.1 Resizing `anon` and `file` memory pools . . . 44
3.4.2 Resizing cgroup memory pools . . . 46
3.5 Conclusion . . . 50

4 Isolation flaws at consolidation 51
4.1 Modeling Consolidation . . . 51
4.1.1 Model assumptions . . . 51
4.1.2 Countermeasures . . . 53
4.1.3 Industrial Application at Magency . . . 54
4.2 Consolidation: once a solution, now a problem . . . 54
4.2.1 Consolidation with containers . . . 54
4.2.2 Consolidation without containers . . . 56
4.2.3 Measuring consolidation errors . . . 56
4.3 Lesson learned . . . 59

5 Capturing activity shifts 61
5.1 *Rotate ratio*: a `lru` dependent metric . . . 61
5.1.1 Detecting I/O patterns that waste memory with RR . . . 62
5.1.2 Balancing `anon` and `file` memory with RR . . . 64
5.1.3 RR can produce false negatives . . . 64
5.1.4 Additional `force_scan`s cost CPU time and impact isolation . . . 64
5.1.5 Conclusion . . . 67
5.2 *Idle ratio*: a `lru` independent metric . . . 67
5.2.1 IR accurately monitors the set of idle pages . . . 68
5.2.2 Trade-offs between CPU time cost and IR's accuracy . . . 69
5.2.3 Conclusion . . . 70
5.3 Conclusion . . . 70

6 Sustaining isolation of cgroups 71
6.1 Refreshing the `lrus` with `force_scan` . . . 71
6.1.1 Conclusion . . . 73
6.2 Building `opt`: a relaxed optimal solution . . . 73
6.2.1 Applying `soft_limits` at all levels of the hierarchy . . . 74
6.2.2 Order cgroups by activity levels with `reclaim_order` . . . 75
6.2.3 Stacking generic policies . . . 76
6.3 Guessing the activity levels . . . 78
6.3.1 A Metric-driven approach to predict activities . . . 78
6.3.2 An Event-driven approach to react to activity changes . . . 80
6.4 Conclusion . . . 82

7 Evaluation of the metric and the event-driven approaches 83
7.1 Experimental setup . . . 83
7.1.1 Workload's types and inactivity models . . . 83
7.1.2 Schedule of activity shifts and configuration of resources . . . 85
7.1.3 Throttling issues with Blkio cgroup . . . 85
7.1.4 Experimental configurations . . . 86
7.2 Performance analysis . . . 89
7.2.1 Control Experiments . . . 89
7.2.2 Event-based solutions . . . 89
7.2.3 Metric-based solutions . . . 90
7.3 Page transfer analysis . . . 90
7.3.1 *Rotate ratio* solutions . . . 92
7.3.2 *Idle ratio* solutions . . . 92
7.3.3 Event-based solutions . . . 92
7.4 Conclusion . . . 93

8 Conclusion and Future works 95
8.1 Short-term challenges . . . 96
8.1.1 Spreading contention on the most inactive containers . . . 97
8.1.2 Ensuring properties when all containers are active . . . 97
8.2 Long-term perspectives . . . 98
8.2.1 Ensuring isolation and consolidation at the scale of a cluster . . . 98
8.2.2 Maximizing global performance with limited memory . . . 99

Bibliography 101
Academical references . . . 101
Technical references . . . 107

Index 115

CHAPTER 1

INTRODUCTION

Cloud computing is now the heart of the digital industry [6]. It cleared the need to own and operate physical hardware by offering remote products to anyone: from web service providers to HPC scientists, from smartphone users to video game players [53, 68, 69, 79]. This transition from a purchasing model to a rental model has been made possible largely through the virtualization of resources.

There are many reasons leading clients to appreciate virtual resources over physical ones. Clients can rent virtual resources on demand with or without time commitment, and they can automatically adjust the number of virtual instances rented according to their load [78, 57]. Once bought, a virtual resource is immediately accessible and does not require shipping. The deployment of common software on fresh virtual resources is optimized in the Cloud thanks to proxy caches or pre-defined disk images. The boot time of virtual resources is also fast because machines in production are rarely shutdown. In addition, virtual resources can be migrated to remain available in the event of hardware failure or maintenance, and to stay up to date when new hardware is available. Devops engineer G. Klok modeled that renting 100 *m4.2xlarge* instances on Amazon Web Service is cheaper than building an equivalent cluster in a carrier hotel [143].

Cloud providers are also far from regretting the days when "bare metal" hardware was the only product to rent. Indeed, once a physical resource has been allocated to a client, it is almost impossible to use that resource for another purpose. Virtual resources are more flexible and they enable such resource multiplexing. This property led to a controversial practice in the Cloud: the sale of more virtual resources than there are physically available. Even if overbooking is not widely advised, it has become necessary for the Cloud. In other industries, it is a common practice used to avoid waste. When buyers are expected to partially consume their entitled resources, overselling ensures that all available resources will be used [76]. Cloud providers gamble on this fact. Thus, they need tools to detect and

consolidate resources within the bounds of the guaranteed isolation level. For example, Amazon Web Service offers spot instances while Google Cloud Platform offers preemptible instances. These products are supposed to sell off unsold resources but nothing forbids Cloud providers to sell off unused resources.

Traditional hypervisors-based virtualization technics are able to efficiently consolidate vCPUs, but they still struggle at transferring unused memory quickly between virtual machines. The heart of the matter is the absence of cooperation between the two entities. Some techniques to consolidate memory have been proposed (Ballooning [84], PUMA [30]), but they are hard to implement and often intrusive (modification of the guest OS), slow (synchronization is required between virtual machines) and manual (the automation proposals are still prototypes).

More recently, lightweight container-based virtualization solutions have begun to emerge and many are now wondering if containers are replacing virtual machines [120]. Even if virtual machines are less prone to security attacks than containers, the latter excel at providing both performance isolation and consolidation because they share the same kernel. In short, performance isolation is what clients care about: they expect virtual resources to perform as well as dedicated bare metal resources. Consolidation is what providers care about: they expect virtual resources to be automatically transferred to clients who truly use them.

My Ph.D. thesis, sponsored by Magency [73] and ANRT [56], studies the full extent of memory consolidation and performance isolation between Linux containers. Despite all the innovative promises of containers, Magency faced a problem when they attempted to use containers to improve resource utilization. Their most active applications encountered momentary slowdowns because they were running out of memory. Yet much of the memory was unused because it was allocated to other inactive applications. In short, we discovered that Linux containers fail at transferring memory from where it is unused to where it is required. Indeed, the need for isolation inside the structures of memory management has deprived the Linux kernel of its ability to correctly identify the most unused memory on the machine. The challenge undertaken by this Ph.D. thesis is twofold. On the first hand, it has to make the kernel aware that there are containers where little memory is used, and containers where more memory is required. On the other hand, it also has to give the kernel the ability to correctly identify the aforementioned containers. Meeting this challenge would allow the kernel to truly offer both consolidation and performance isolation. The contributions of this research are:

- A simple experiment that highlights the problem with real applications used by the Google Perfkit [66] (MySQL [75] and Cassandra [58]) that could occur in a production environment.

- A synthesis of a careful code analysis of the Linux kernel, grasping the root of the problem that resides inside the structures of memory management.

- The design and implementation of two kernel-level approaches that strive to detect which containers have too much memory and which containers have too little memory. Both approaches are aware that memory needs of containers must be ordered, but the first approach relies on metrics to make that distinction, while the other approach relies on kernel events.
- A thorough evaluation, using two types of workload (Sysbench [144] and Memtier [74]) under two activity models, that pushes both approaches to their limits.

This document is organized as follows: *Chapter* 2 presents the two technologies that virtualize resources, *Chapter* 3 presents the technical background required to understand why Linux struggles at correctly transferring memory between containers, *Chapter* 4 demonstrates this problem with a simple experiment that could occur in production, *Chapter* 5 studies two kernel metrics that can detect when memory is unused, *Chapter* 6 proposes kernel modifications to preserve isolation during consolidation, *Chapter* 7 evaluates our solutions, and *Chapter* 8 concludes and presents future works. In-depth, the chapters are as follows:

Resource virtualization is the Cloud's cornerstone but, *Chapter* 2 suggests that for efficiency reasons, some workloads deployed in virtual machines should shift to containers. The first part praises the properties of virtual machines that gave birth to the Cloud. However, by design, hardware-level virtualization introduced additional costs that limit the efficiency of the Cloud. In the second part, *Chapter* 2 demystifies the container's recipe, which is no more than a mixture of Linux kernel features that were developed separately over time to meet the different needs for isolation. The third part presents the literature work that compares containers to virtual machines and concludes that the former are more efficient than the latter. The fourth part introduces and illustrates through tangible experiments the two core concepts of this thesis: performance isolation and consolidation. While both of these properties can be ensured in the case of CPU and disk bandwidth, the very last part of this chapter shows that the properties are harder to ensure in the case of memory.

To fully understand the problem addressed in this thesis, *Chapter* 3 dives into some basics of **memory and cgroup** in the Linux kernel. The first part presents memory as a resource that is managed in page units and that can be spatially and temporally multiplexed. The second part details how memory is accounted for and how it is limited in the Linux Kernel. The third part explains how Linux stores pages in pools to order their utility, and then explains why they cannot be stored in a single pool. The last part explains how the memory pools of the same container are dynamically resized, but as the relative works concede, the same principles cannot be applied with pools of different containers.

The aforementioned problem is hard to solve in general, but *Chapter* 4 suggests that some **isolation flaws at consolidation** can be prevented. The first part presents a specific model of memory activity where isolation should be sustained, and how this model was business inspired by the applications of Magency [73]. The second part reproduces the problem encountered at Magency with a simple experiment. Additionally, it shows through a micro-benchmark that workloads which do not require isolation are correctly consolidated when

deployed without containers, but incorrectly consolidated when deployed within containers. Finally, the last part applies the knowledge acquired in *Chapter* 3 to explain the results observed in these experiments.

The first step towards resolving the problem is to **capture the activity shifts** with memory-related metrics. In *Chapter* 5, two metrics are studied: the *rotate ratio* and the *idle ratio*. Measurements of their cost and accuracy are provided in a context where activity changes but without the intent of consolidating memory.

Chapter 6 presents the modifications we developed in the kernel to **sustain the isolation of cgroups** during consolidation. The first mechanism developed allows the user to control the freshness of the *rotate ratio*. In the second part, a series of mechanisms are described. They enable generic memory reclaim policies to be expressed and in particular, they enable a relatively optimal solution to be expressed. In the last part, two classes of solutions are proposed. Their goal is to guess the activity levels as accurately as the optimal solution: one class is based on the metrics presented in *Chapter* 5, and the other one is based on kernel events which are "page demands" and "page activations".

An **evaluation** of these solutions is reported in *Chapter* 7. The first part presents the methodology of our experimental setup which uses two workload types with two inactivity models. These experiments are harder to consolidate than the ones presented in *Chapter* 4. The remaining parts report the performance and the consolidation errors of the solutions with respect to the least performing container. The evaluation highlights the strengths and weaknesses of the different solutions and shows their limit compared to the optimal solution that perfectly knows the activity levels.

Chapter 8 **concludes** and presents **future works** and **perspectives**. It summarizes the work done and highlights some of the experimental results that we were able to achieve. On a short-term, we anticipate contentions with our prototypes as the number of container increases. Then, we provide some leads on how to extend our work to the context of overcommitment, when all containers become active. Finally, we suggest that the metrics and events studied in this thesis should also be studied at the scale of the cluster to unlock the challenges of node balance, boot up and shut down.

CHAPTER 2

RESOURCE VIRTUALIZATION

In this chapter, we present the virtualization techniques employed in the Cloud, and how they are used to increase the yield of the machines.

Hardware-level virtualization introduced key properties that gave birth to the Cloud, but more recently, Operating System-level virtualization has begun to emerge as a lightweight alternative that offers similar properties.

2.1 Virtual Machines: Hardware-level virtualization

Hardware-level virtualization was essentially achieved by introducing a hypervisor: a new software layer between a guest operating system and the hosting hardware. As A. Tanenbaum explains it, there are two types of hypervisors [45]: type 1 hypervisors run on bare

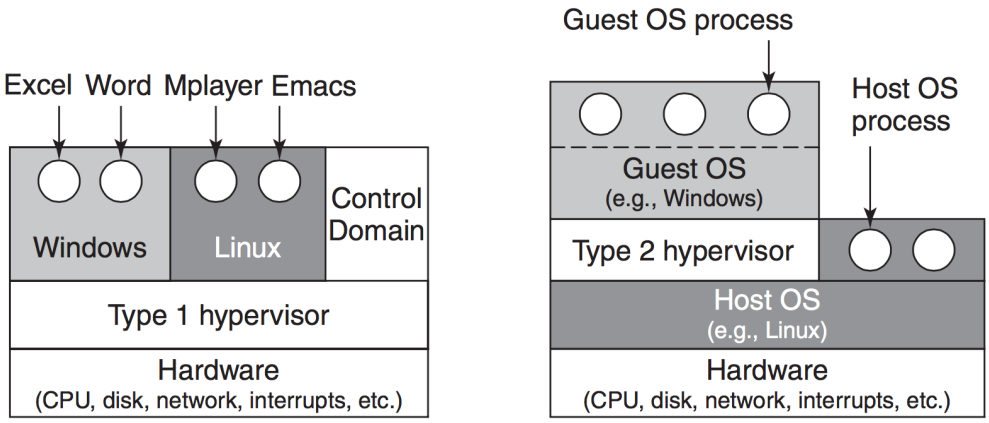


Figure 2.1: Hypervisor types.

metal, and type 2 hypervisors make use of the abstractions offered by an underlying operating system (OS) (see Figure 2.1). Either way, the goal of the hypervisor is to host multiple virtual machines (VMs) on a single computer. The VMs are accurate, isolated and efficient duplicates of the real machine [38]. Several interesting properties emerged from the use of VMs:

1. **Emulation**: Hypervisors create the illusion that the VM is in charge of the hardware. This ability to host any OS separately leads to many applications ranging from running legacy software to running OS-exclusive software, and from debugging kernel development to debugging multi-OS software development.

2. **Safety**: Hypervisors are less prone to bugs than OS since they do one thing exclusively: emulate multiple copies of the bare metal. Errors and failures are not likely to propagate from VMs to VMs or to the hypervisor.

3. **Security**: Hypervisors usually do not allow multiple VMs to access a given physical resource at the same time. The VM-level attack surface is small compared to the process level. Unfortunately, both surfaces are sensitive to hardware designs [28, 26].

4. **Economy**: Hypervisors save money on hardware, electricity and rack space in data centers because fewer physical machines are needed when single machines are multiplexed into multiple VMs.

Companies specialized in data center management and staffed by experts in the area took advantage of these properties; and gave birth to the Cloud by allowing clients to remotely access their physical resources through virtualization. VMs in the Cloud are undeniably appealing to clients because in contrast to physical machines, they are resizable, already powered, cooled, maintained and upgraded by the provider. On one hand, economies of scale are achieved by deploying multiple clients on the same machine, but on the other hand, their privacies and the quality of their services are put at risk.

2.1.1 Cost of machine virtualization

Tremendous efforts have been made to improve as much as possible the efficiency of machine virtualization. Prior to hardware-assisted virtualization, the trap-and-emulate technique used to prevent a VM from executing sensitive instructions was not enough, and hypervisors had to dynamically translate these instructions. Today, thanks to hardware-assisted features (Intel VT-x, AMD-V), the cost of machine virtualization is now acceptable: *i)* CPU-wise, additional instructions are incorporated to meet the formal requirements[1] of Popek and Goldberg [38]. *ii)* Memory-wise, additional registers are incorporated to let the MMU access the nested page tables needed for the double address translation (Adams and Agesen [1]). *iii)* IO-wise, the addition of an IOMMU can provide device isolation [9], and

[1] All sensitive instructions (that can affect the hypervisor) must be privileged instructions (that can be trapped by the hypervisor).

SR-IOV devices can now appear as multiple separate devices [15]. Unfortunately, it has been reported that despite being small on machines with few cores, the overhead becomes unacceptable on large NUMA machines [48]. Moreover, VMs do not maximize resource utilization and countless dedicated schemes had to be conceived to tackle this limitation.

2.1.2 Improving utilization in a full virtualization context

Very few schemes respect the full virtualization paradigm in which VMs are non-cooperative black boxes that cannot be modified. The most common scheme targets the duplication of identical data caused by the deployment of multiple instances of the same guest OS. Thanks to works such as Linux KSM [5], the hypervisor strives to de-duplicate data to save memory. VSwapper [3] is another scheme that respects full virtualization and tries to address the double paging anomaly [20]. This anomaly occurs when both the hypervisor and the VM are running eviction policies that end up contradicting themselves. Goldberg showed that an increase in the size of the memory of the virtual machine without a corresponding increase in its real memory size can lead to a significant increase in the number of page faults. Memory hot-plug emulation by the hypervisor was also suggested as a means of dynamically balancing memory between VMs [127].

2.1.3 Improving utilization in a paravirtualization context

The remaining majority of the schemes fall into the paravirtualization domain because the absence of cooperation between the VM and the hypervisor creates too much complexity. The most extreme form of paravirtualization requires the guest OS to be explicitly ported to communicate with the hypervisor through the use of hypercalls. At the other end of paravirtualization, some less intrusive schemes take advantage of existing interfaces in the guest OS to insert additional communication logic. Ballooning [46, 84] uses a Linux virtio driver [41] and allows the hypervisor to ask the guest to free its memory. PUMA [30, 29] uses the `Cleancache` API of Linux [90] and allows a VM to lend its unused memory to another remote VM.

To sum up, machine virtualization ends up increasing execution time and memory space, and burdens software and hardware development. Fortunately, containers are less subject to these drawbacks and open the door to a more efficient Cloud.

2.2 Containers: Operating System-level virtualization

The virtualization property the most requested by lambda users is the ability to encapsulate and run anywhere an entire environment, including software dependencies, libraries, a runtime code, and data. Singularity is one of the container engines that solely focus on this idea [27], but most of the container engines take advantage of all kernel features available to enforce isolation. There is no such thing as a container kernel object [104]. Therefore, one can define a container as an assembly of kernel isolation features. For example, Docker

Cgroup	Resource
`cpu`	time on CPU
`cpuset`	CPU cores and memory nodes
`memory`	ram usage
`blkio`	block input output
`freezer`	pause/unpause
`devices`	open/mknod on device files
`pids`	number of process
`hugetlb`	Huge page (2 MB, 1 GB)
`net_prio`	priorities for queuing disciplines on network packets.
`net_cls`	filter and classify network packets.
`perf_event`	aggregate perf utility events.

Table 2.1: Cgroup types and related resources.

sells its container engine as a solution to "Build, Ship, and Run Any App, Anywhere" [63], but in the background, it makes use of cgroups, namespaces and security features.

2.2.1 Isolating physical resources with cgroups

Prior to cgroups, utilities such as *nice*, *ionice*, *mlock*, *madvise*, *fadvise*, *taskset*, *numactl*, *trickle* [16] and *setrlimit* could be used to control a single process, but no such things existed to control a group of processes.

Cgroup stands for "control group". It is a Linux Kernel feature that groups processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner [147, 125]. Containers use cgroups to **limit**, **account** for, and **isolate** the physical resource usage. As N. Brown explains [82], there have been some disagreements on considering this grouping of processes as an organization hierarchy or as a classification hierarchy, but both views are correct. In a classification hierarchy, all members cannot be in internal nodes, but in an organization hierarchy, members in charge of managing others are placed inside internal nodes. The current API version of cgroups is very messy and inconsistent across resources, but these issues are going to be fixed in version 2 [126, 150]. The cgroup API is exposed as a virtual filesystem mounted at `/sys/fs/cgroup` and processes can look up their membership at `/proc/PID/cgroup`.

The remaining of this subsection details the cgroup subsystems listed in Table 2.1. As cgroups are still under development, additional cgroup subsystems will be added such as the `rdma` cgroup [149] and the memory bank and CPU cache cgroup [51]. More cgroup subsystems could be conceived; for instance, it could be wise to implement a cgroup that controls the memory bandwidth [11].

The `cpu` cgroup

The `cpu` cgroup has two major features. The first feature statically throttles tasks. It can be seen as a CPU quota or a CPU bandwidth. It has two parameters: one defines how much time tasks can run on the CPU and the other defines the period of time required to renew the former running quota. The second feature dynamically throttles tasks when there is no idle CPU time left. It has a sharing parameter that guarantees that the tasks will at least be able to acquire an amount of CPU time proportional to the parameter[2]. Both features complement each other. The first helps to avoid CPU shortage, while the other one dictates guidelines to the Completely Fair Scheduler (CFS) when it has to handle CPU shortages.

The `cpuset` cgroup

The `cpuset` feature predates the generic cgroup implementation and was the first one to be added. It restricts processes to run on a subset of CPUs and forces them to allocate memory on a subset of memory nodes. The `cpuset` cgroup offers many more features [108]: cores and nodes can become exclusive, memory pressure can be monitored, memory allocations can be spread on all nodes instead of preferring the node where the allocations were made, load balancing can be partitioned into domains, etc.

The `memory` cgroup

The `memory` cgroup subsystem, which is at the heart of this thesis, is thoroughly detailed in Chapter 3. In a nutshell, it throttles tasks when they attempt to exceed the memory limit of their cgroup. Indeed, as they reach the limit, the tasks are forced to run the Page Frame Reclaiming Algorithm which recycles some of their existing memory. The `memory` cgroup also controls the memory consumption of kernel objects, the swap and TCP buffers.

The `blkio` cgroup

The `blkio` cgroup is similar to the `cpu` cgroup. It can statically throttle reads and writes per devices in terms of bytes per second or in terms of operations per second. It also has weight parameters per devices that dynamically guide the Completely Fair Queuing (CFQ) I/O Scheduler when the devices are saturated[3].

The `freezer` cgroup

The freezing feature was originally developed for hibernation and is more powerful than the *SIGSTOP*/*SIGCONT* mechanism because it cannot be caught by processes. The `freezer` cgroup can be used to schedule batched jobs from userspace [109, 65, 157], but can also be used for checkpointing and migration with CRIU [62]. As the user freezes a cgroup, a fake signal is sent to all its processes but also to those of its children. When it receives a

[2]The sharing parameter is not available with the Real-Time Scheduler (R-T).

[3]Provided that CFQ is enabled for the device (`/sys/block/device/queue/scheduler`).

signal[4], a process will go to the refrigerator [5] if its cgroup status is frozen. Moreover, newly created processes cannot escape the freezing mechanism because the `freezer` cgroup gets notified when a process forks. As the user thaws the cgroup, processes are woken up and leave the refrigerator.

The `devices` cgroup

The `devices` cgroup provides mandatory access control (MAC) to block and character device files. It takes into account the hierarchy: children do not have more permissions than their parent. When the user updates the permissions of a cgroup[6], *i)* new restrictions are always propagated to its children, and *ii)* new authorizations are requested to its parents who can deny them. The permission checks are therefore relatively faster than their updates because the hierarchy does not have to be walked through. Read, write and create permissions per cgroup are checked through two new kernel functions[7] when a process calls `open` and `mknod`.

The `pids` cgroup

The `pids` cgroup controls the `fork` system call and prevents a new process from being created if its cgroup has more processes than its maximum limit. As the accounting is hierarchical, when a child cgroup or any of its parent reaches their limit, the `fork` system call fails by returning the "try again" error (`-EAGAIN`). At first glance, processes can be seen as kernel structures in memory and therefore, limiting the memory consumption of kernel objects that belong to a cgroup should, in theory, prevent process identifiers (pids) from being exhausted. But in practice, the total number of pids in the system is currently bounded[8] to 2^{22}. Therefore, without the `pids` cgroup, a fork bomb can easily exhaust the `pid` table without hitting its kernel object memory limit.

The `hugetlb` cgroup

Huge pages are used to minimize the entries to look up in the Page Table and therefore minimize TLB misses (translation look-aside buffer). Once allocated to a global pool, preferably at boot time, huge pages cannot be reused for another purpose and cannot be swapped out. The `hugetlb` cgroup controls the number of huge pages that cgroups are allowed to allocate.

The `net_cls` and `net_prio` cgroups

The network cgroups (`net_cls` and `net_prio`) provide means to identify packet ownership and means to override packet priorities, but they do not take advantage of the hierarchy

[4] The function `get_signal` calls `try_to_freeze` before handling pending signals.
[5] The `__refrigerator` is a loop on `schedule`.
[6] The function `devcgroup_update_access` modifies the MAC.
[7] The functions `devcgroup_inode_permission` and `devcgroup_inode_mknod` are called to enforce MAC.
[8] See `/proc/sys/kernel/pid_max`.

Namespace	Isolates	Hierarchical
User	User and group IDs	Yes
PID	Process IDs	Yes
Mount	File system mount points	Propagation rules
Network	Network sockets, devices, tables, etc.	No
IPC	System V IPC, POSIX message queues	No
Cgroup	Cgroup path names, root cgroup	No
UTS	Hostname and NIS domain name	No

Table 2.2: Namespace types and isolated resources.

and do not directly provide limitation, accounting and isolation features yet [102]. The network cgroups extend the following kernel objects: *i)* the network device object has now a priority for each cgroup[9], and *ii)* the socket object has now two extra fields called `classid` and `prioidx`. When a process creates a new socket, the `classid` and `prioidx` fields are initialized with the values of the cgroup from whom it belongs. `prioidx` corresponds to the cgroup index in the priority arrays, while `classid` is set from userspace. `classid` can then be used with *iptables* to selectively filter packets (firewall rules), but can also be used with *tc* (traffic control) to classify packets during network scheduling. Meanwhile, the priority values can be used to override the `SO_PRIORITY` option used by the queuing discipline on packet delivery and delay [61, 60].

The `perf_event` cgroup

The `perf_event` cgroup is the most rudimentary cgroup. It allows *perf* [122], the performance analyzing tool of Linux, to collect and aggregate performance data of processes that belongs to a cgroup. The `perf_event` cgroup uses the hierarchy, therefore metrics from processes that belong to the children of the monitored cgroup are also collected[10].

Conclusion

Cgroups are a well-integrated, stand-alone feature of the Linux kernel. By default, there is a single root cgroup in which processes spawn, but container engines and even common Linux distributions create additional cgroups to isolate resource utilization. However, as we will see in Chapter 4, most users are unaware that creating multiple cgroups can lead to undesired pitfalls [52].

2.2.2 Isolating resource visibility with namespaces

Prior to namespaces, utilities such as *chroot* or *pivot_root* could be used to locally reshape the view of the filesystem, but there were no generic API to instantiate a concurrent view of any given resource.

[9] Priorities are stored in the `netprio_map` array and can be configured from userspace.

[10] `perf`'s `event_filter_match` if the CPU context `cgroup_is_descendant` of the event cgroup.

As M. Kerrisk explains [135], namespaces are a Linux kernel feature that creates private local views of system resources for processes and gives them the illusion that they are the sole set of processes controlling the resources. Containers use namespaces to forbid undesired OS-level interactions between processes. Contrary to cgroups, only two types of namespaces are purely hierarchical: User and PID. Nevertheless, since parent processes have access to the `/proc/PID/ns/` files of their children, the remaining namespace types can still be configured in a hierarchical fashion.

Namespaces can only be created through system calls; they are destroyed if there is no process remaining in the namespace and if there are no more references[11] to their `proc` file. Given the appropriate options, four system calls manipulate namespaces [136, 70]:

- `clone` will create its new process into newly created namespaces.
- `setns` allows processes to join other namespaces previously created.
- `unshare` creates new namespaces and makes the calling process join them.
- `ioctl_ns` exposes the namespace membership and hierarchical relationship through the `/proc/PID/ns/` files.

E. Biederman initially identified ten namespaces [10] and the remaining of this subsection details the most mature namespaces among them (listed in Table 2.2). There are many more namespaces to implement (such as syslog [132], device [112, 4, 146, 59], sysfs [88], security modules [148, 85, 128], keyrings [121], and time [121]), but they must not introduce new vulnerabilities (such as the ones previously reported [100, 97, 96, 134]). Moreover, there are still some unresolved issues with the current namespaces such as unprivileged filesystem mounts [98], file capabilities [105], and uid stored in filesystems [117, 119].

The User namespace

The User namespace [139, 140, 133] remaps the real user (UID) and group (GUID) identifiers to any custom virtual numbers. As these identifiers can be remapped to those of the superuser, the user namespace also provides the illusion that processes acquire full privileged capabilities. But in reality, there is no privilege escalation because these capabilities are only valid for operations on objects created inside the user namespace. Regarding the objects created outside, the capabilities are restricted to the ones that were already granted. Unprivileged processes are allowed to create user namespaces and the other types of namespaces (PID, Mount, Network, IPC, UTS, Cgroup) are said to be **owned** by the user namespace from which they were created. As a result, operations such as changing the root of the filesystem[12], setting the hostname, mounting virtual filesystems[13] and binding reserved socket ports are authorized only if the process has the right capabilities[14] in the user

[11] The `proc` file is nor opened neither mount bound.

[12] `chroot`

[13] `/proc`, `/sys`, `tmpfs`...

[14] `CAP_SYS_CHROOT`, `CAP_SYS_ADMIN`, `CAP_NET_BIND_SERVICE`.

namespace which owns the corresponding non-user namespaces. As User namespaces are nested, processes in a parent namespace can still control resources in the children user namespaces.

The PID namespaces

The PID namespaces [137, 138] are nested and processes have a different identifier (PID) at each level of the hierarchy. From one namespace perspective, a process has a given PID, but from a higher namespace perspective, its PID is different. Due to the hierarchical nature of the PID namespaces, a process only has access—and thus, for instance, can only send signals—to other processes that belong to its namespace, or that belong to a descendant of its namespace. The PID namespace eases migration because processes can keep their PID on the new host even if they are already attributed to some other processes in an ancestor or sibling namespace. The first process to populate a namespace (`init`) is as critical as the first process spawned at boot time (the real machine-wide `init`). In its namespace, `init` has PID 1. As the "ancestor of all processes", `init` is in charge of initialization, reaping of terminated orphaned processes, and graceful termination of the whole namespace[15]. `Init` is so important that at its death, the kernel sends `SIGKILL` to the remaining processes and rejects[16] the creation of new processes, making the namespace unusable[17]. Therefore, signals sent to `init` are ignored if no corresponding handler has been declared; but processes from a parent namespace can still send `SIGKILL` or `SIGSTOP` signals.

The Mount namespace

The Mount namespace [141, 142] provides a private view of the filesystem trees and allows processes to independently reshape their view. When a new mount namespace is created, it duplicates the current view of the filesystem. Afterwards, as each namespace has its own list of mount points, the results of *mount* and *umount* operations in a namespace are not visible to other namespaces. Additional options are also available to cherry-pick mount point modifications and propagate them from namespace to namespace.

The Network namespace

The Network namespace [114, 87] can be used to configure, on one machine, any intricate virtual networks. These networks can allow or deny incoming or outgoing connections between namespaces or with the Internet. Each namespace has its own network-related resources such as sockets, addresses, interfaces, routing tables, firewall rules, etc. When a new network namespace is created, it only has a `loopback` device. As network interfaces can only belong to one namespace, the system will run out of physical interfaces. Therefore, pairs of virtual interfaces can be created and moved between namespaces and with

[15] `docker stop` sends `SIGTERM` before `SIGKILL` to the `init` process of the container.
[16] `fork`, called after some `setns` or the `unshare` that created the namespace, returns `ENOMEM`.
[17] Processes cannot join another PID namespace, their membership is defined at creation.

the addition of some network configurations, the newly created namespace can, for example, have access to the Internet.

The IPC namespace

Processes belonging to different IPC namespaces [145] cannot share interprocess communication objects such as System V IPC and POSIX message queues because each IPC namespace has its own System V IPC identifiers[18] and its own POSIX message queue filesystem[19]. The IPC namespaces are not hierarchical.

The Cgroup namespace

The cgroup namespace [113] hides the full cgroup path from the global cgroup hierarchy and gives the illusion that contained processes are at the root cgroup. The cgroup namespace eases process migration and allows container-management tools to be nested.

The UTS namespace

The UTS namespace [136, 124] hides the hostname and the NIS domain name, and provides private system identifiers to the contained processes. By default, Docker labels the hostname with the container identifier.

Conclusion

Namespaces are, just as cgroups, a well-integrated feature of the Linux kernel. By default, there is a single root namespace in which process spawn, but container engines and even web browsers create additional namespaces to isolate the visibility of resources. However the visibility of resources is not the subject of this thesis and, as such, when we refer to isolation in the rest of this manuscript, we will not consider its visibility aspect.

2.2.3 Restraining attack surface with security features

The principle of least privilege requires that an application must not be able to access information and resources other than the ones that are necessary for its legitimate purpose [42]. In addition to cgroups and namespaces, container engines rely on Linux security features to apply this principle. Moreover, they also rely on security features to deny access to resources that have not yet been fully isolated with cgroups or namespaces.

The remaining of this subsection details Linux Capabilities, Secure Computing and Security Modules, but in general, any systemwide hardening technic such as applying patches [67] is compatible with containers.

[18] `ipcs` will not list identifiers from another IPC namespace.

[19] `mount -t mqueue none /dir` will not contain message queues from another IPC namespace.

Linux Capabilities

Linux Capabilities is a security feature that divided all the privilege of root into smaller distinct privileges called capabilities [21]. Prior to capabilities, the `setuid` permission, on an executable file of root, allowed normal users to run the program with all the privileges of root. Today, if a privileged program is compromised, the damage that it can do is limited by its capabilities. Unfortunately, a lot of privileges fall into the `cap_sys_admin` capability which tends to become the new full-privileged mode [129].

By default, Docker runs containers with a restricted set of 14 capabilities over the 38 available. `cap_net_bind_service` is for instance granted to allow processes like web servers to bind on a port below 1024; `cap_sys_module` is for instance denied because it would allow corrupted containers to insert a rootkit module. Docker users can adjust this profile to suit their security needs [20].

Secure Computing or `seccomp`

`seccomp` is a security feature that prevents processes from interacting with the kernel. In its most restricted form, `seccomp` will kill a process if it attempts to execute any system call other than `exit`, `sigreturn`, `read` and `write` to file descriptors opened beforehand. It was originally designed by A. Arcangeli to securely rent out CPUs with Linux [86], but Google hijacked the idea to securely run plugins in its Chrome browser, even if at the time, four system calls were too restrictive [89]. Later on, W. Drewry borrowed a network filtering feature to enhance the flexibility of `seccomp` [94]. As J. Edge explains [116], sandbox developers can now write complex filters on system calls and their arguments, using a mini-program in the Berkeley Packet Filter language (BPF [35, 95]) which is verified and compiled by the kernel. Moreover, with `seccomp-bpf`, when a process violates the policy, several outcomes are available:

- Killing the process.
- Sending the process a `SIGSYS` signal.
- Failing the system call and returning a (filter-provided) errno value.
- Notifying an attached process tracer (provided that one is attached with `ptrace`[21]).
- Allowing the system call.

All container engines and sandboxing software now use `seccomp`. Docker, for instance, disables around 44 system calls out of 300+, including `mount`, `reboot` and `setns`.

[20] The `--cap-add` and `--cap-drop` options adjust the Linux capabilities of a Docker container.

[21] The process tracer can skip or change the system call.

Linux Security Modules (LSM)

Prior to LSM [49], the mainstream Linux kernel provided discretionary access controls (DAC) that only check if the identity or group of the subject matches those of the object. Unfortunately, DAC blurs the difference between users and applications because users have to pass their permissions to applications. As a result, LSM was developed as a lightweight, general-purpose framework that enables many access control models[22] to be implemented as loadable kernel modules.

Most of the kernel hooks provided by LSM are restrictive; they allow security modules to overload the DAC checks. In other words, these hooks are not triggered if the DAC policy already denied the access. However, there are some exceptional permissive hooks that override the DAC checks, but these were only added to support the logic of Linux Capabilities. Moreover, LSM inserts a pointer in kernel objects that allows security models to bind attributes to the objects. Modules that use these blobs are called major modules and the others are called minor. At this point, stacking of minor modules is supported, but there can only be one major module at a time since there is only one security-blob pointer per object [103].

Major stacking is still under development [115, 111, 39] and other works are trying to make LSM namespaced-aware [148, 85, 128]. By default, templates for Docker are provided for the most popular LSMs which are `SELinux` [31, 110] and `AppArmor` [13, 55].

Conclusion

Security features are more or less integrated by default into the Linux kernel. Containers engines and even web browsers take advantage of these features to harden the sandboxes in which distrusted programs are executed. However security is not the subject of this thesis and as such, when we refer to isolation in the rest of this manuscript, we will not consider its security aspect.

Now that we have demystified the major internal components of containers, we will compare them to VMs in the next section.

2.3 Containers and VMs comparison

G. Costa once “heard that hypervisors are the living proof of operating system’s incompetence” [131]. Indeed, operating systems were originally designed to greedily exploit every ounce of resources physically available, in the hope of improving the overall performance (blocked tasks are rescheduled, free memory caches disk pages, unaccessed pages are swapped out of memory...). In this model, the machine is assumed to be dedicated to a single application, and fairness is enforced between its tasks to ensure that they make progress (CFS, CFQ, Swap Token...). Unfortunately, at the time, we did not envision that machines would be powerful enough to host multiple applications at once. As a result,

[22] Such as mandatory access control (MAC).

services that in theory could hold on the same machine were deployed into separate physical machines because Operating Systems did not provide isolation at the application level. Eventually, this practice lead to the fragmentation of idle resources which in turn became one of the incentives to develop hypervisors. The bottom-up approach of VMs was the easiest and fastest way to provide isolation, but today, this habit of sharing almost nothing between virtual instances comes at a burdening price, easily outmatched by the top-down approach of containers.

2.3.1 Comparing stand-alone overheads

There are many works in the literature that compare the stand-alone virtualization overhead of VMs and Containers [37, 44, 40, 50], but since this technology evolves fast, we will focus on the recent work done in 2015 by W. Felter et al. in which they concluded that containers result in equal or better performance than VMs in almost all cases [17].

One property of hypervisors is to abstract and hide the underlying physical resources, but W. Felter et al. showed that it also eliminates some opportunities for optimization. As a result, their Linpack benchmark was only able to execute 125 GFLOPS in KVM compared to 275 GFLOPS when running in Docker.

Hypervisors require an extra hardware page table walk to handle TLB misses, but W. Felter et al. showed that it can become a bottleneck on a single CPU socket. As a result, their random memory access benchmark was only able to execute 0.04G updates/s in KVM compared to 0.045G updates/s when running in Docker.

Paravirtualization technics such as `virtio` are often employed to minimize virtualization overhead, but W. Felter et al. showed that KVM delivers only half as many IOPS as Docker. Moreover, KVM's read latency is two to three times higher than Docker's.

To test the network overhead, W. Felter et al. used Redis. Surprisingly, KVM was able to outcompete Docker when the number of clients is greater than 30. It appears that when Docker uses NAT, it introduces latencies that grow with concurrent connections. On the other hand, KVM initially has more latency than Docker, but as concurrency increases, it is able to fully utilize the system.

As a final evaluation, W. Felter et al. used MySQL and Sysbench. They showed that Docker had similar performance to an unvirtualized environment, within a difference of 2% at higher concurrency. On the other hand, KVM had higher overhead, higher than 40% in all measured cases.

2.3.2 Comparing performance isolation and overcommitment

In 2016, P. Sharma et al. observed the same results obtained by W. Felter et al., but they also conducted experiments to evaluate the performance isolation of VMs and Containers when a "noisy neighbor" is executing beside them [43].

They first compared LXC to KVM in terms of CPU confinement: with their competing scenario[23], LXC and VM obtain the same results. With their orthogonal scenario[24], they observed that `cpushares` on its own results in a greater amount of interference, up to 60% higher compared to the baseline case of stand-alone no-interference performance. We believe that this might be due to load balancing bugs in the Linux scheduler [32]. Finally, their fork-bomb adversarial scenario is outdated, because the cgroup pid was introduced in 2016.

They claimed that memory isolation provided by containers and VMs are sufficient for most uses but they suggest that containers could suffer more from a malloc-bomb adversarial scenario. We believe that this case might be wise to investigate since kernel structures, in charge of free memory, are indeed not isolated.

In terms of disk confinement, they observed a reduction of 8x in the case of containers with their competing scenario, but they did not provide any explanations. For instance, was the competing workload consuming more bandwidth than monitored workload? Was the Completely Fair Queuing I/O Scheduler enabled on the device [25]? Since rotating disks do not serve concurrent requests as well as solid-state disks (SSDs), we ask ourselves if data were correctly placed on disk. Moreover, they only tested the weight cgroup feature and did not test the fixed bandwidth or IOPS quota feature.

P. Sharma et al. also compared the overcommitment potential of VMs and Containers. They showed that VM performance is within 1% of LXC performance when the CPU is oversubscribed by a factor of 1.5. On the other hand, when memory is oversubscribed by a factor of 1.5, the VM performed about 10% worse than LXC. Finally, by using memory `soft_limit`, they showed that the VM performed about 40% worse than LXC with an oversubscribed factor of 2.

2.3.3 Should VMs mimic Containers?

Aside from wondering if containers are replacing VMs, some works are now pursuing the goal of making VMs as efficient as containers.

The major argument in favor of VM has always been security [118]. Indeed, the Linux syscall API is growing and is more difficult to secure than the static x86 ABI with its CPU protection rings. Lightweight VMs have been suggested as an efficient alternative to classic VMs and as a safer alternative to classic containers. In 2017, F. Manco et al. were able to achieve container-like properties with LightVM [34]: *i)* Fast instantiation, *ii)* High Instance Density and *iii)* Pause/unpause.

They observed that the size of the guest VM is one of the biggest limiting factors, and therefore suggested two approaches to minimize the VM footprint: the most compact approach

[23] The noisy neighbor is CPU intensive.

[24] The noisy neighbor is CPU and memory intensive.

[25] The default scheduler could be `deadline` which does not enforce fairness between cgroups.

is to make use of unikernels which unifies the kernel and the user application into a single binary. However, they conceded that linking existing applications that rely on syscalls to an unikernel such as Mini-OS requires a lot of expert time. Therefore, they suggested a less compact approach called Tinyx which automates the creation of minimalist Linux VM images.

In addition to shrinking down the size of the VM, F. Manco et al. also needed to re-engineer an existing hypervisor to achieve boot times comparable to `fork/exec` on Linux. They picked Xen, replaced the message passing protocol of xenstore with shared memory, sped up the creation of VMs with templates, and named the result LightVM. Their boot time evaluation shows that unikernel over LightVM is in the order of 10 ms whereas Docker and Tinyx over LightVM are in the order of 100 ms. In terms of CPU usage and memory footprint, unikernel over LightVM achieves similar consumption than Docker, and Tinyx has an overhead negligible compared to classic Debian VM.

However, containers possess another very useful property, more important than the aforementioned ones, that this work did not try to achieve with VMs. For example, stateful services such as databases do not need fast instantiation because when their load increases, it is wiser to scale up their current capacity rather than booting up new instances on the same machine. Unlike containers, VMs cannot provide such flexibility because underlying physical resources were never meant to change at runtime. In terms of security, instead of adapting applications to unikernels, expert time can be spent to narrow down the attack surface of the syscall API with `seccomp`. Moreover, this procedure can be sped up thanks to automated systems that audit and profile application interactions with the kernel.

To conclude, VMs were developed because there were use cases that OS could not handle due to their lack of isolation features. Today, most of these use cases can be handled by containers, the sole exception being running another kernel. In the future, the security of containers will mature, but if VMs are to shrink down to the size of processes and start to strongly cooperate with their hypervisor to extend their flexibility, it would be like reinventing the wheel.

2.4 Consolidation and Isolation, the best of both worlds

Many aspects of virtualization have been discussed so far, but in the remainder of this thesis, we will focus on virtualization as a means of providing more virtual resources than there are physically available. Aspects such as encapsulation or security will be put aside; and since VMs are less efficient in terms of resource yield, we will only focus on containers.

Consolidation and isolation are key concepts to study when resources are multiplexed. In short, performance isolation is what Cloud clients care about; whereas resource consolidation is what Cloud providers care about.

2.4.1 Resource Consolidation

Opportunities to multiplex resources exist in the Cloud. Indeed, R. Ghosh observed in an internal private Cloud that: *i)* 84% of the running VMs have a CPU utilization peak less than 20% and *ii)* only 0.7% of VMs have a maximum CPU utilization close to 100% [19]. Moreover, these opportunities are not only spatial (because multiple small utilization peaks can be packed together), but there are also temporal because utilization peaks do not occur at the same time.

Resource multiplexing opportunities are sometimes hard to exploit, but there are ways to unveil them by reshaping the workloads. For instance, P. Lu et al. designed a Hadoop scheduler that colocates MapReduce tasks to maximize resource utilization and minimize resource contention [33]. They were able to improve the CPU utilization by 10%, halve the I/O wait time and reduce the job execution time by 10%.

To conclude, consolidation is about taking advantage of unused resources to increase the yield of machines in the Cloud or to reduce the job execution time.

2.4.2 Performance Isolation

The need for performance isolation emerged and grew when systems started to handle multiple tasks and users [47]. Ideally, tasks running on the same machine should behave as if they were running alone on separate machines. In theory, performance isolation is achievable given enough resources on a single machine. Moreover, even if there are not enough resources for all the tasks, a subset of tasks should be able to behave as if they were running alone. For instance, highly interactive user-end tasks[26] usually do not consume many resources; and as a result, their performances are easier to preserve even if the machine is under heavy loads. But heavy applications tend to compete for resources and do not take into account possible impacts on other applications in the system when they ask for resources. Instead, the resource management logic is decoupled from the application to ease software development and delegated to the kernel to provide flexibility at runtime.

To conclude, isolation is about providing virtual resources that perform as well as dedicated bare metal resources.

2.4.3 Illustrating Consolidation and Isolation with the CPU cgroups

To illustrate the concepts of consolidation and isolation, we crafted an experiment involving two MySQL servers (called *A* and *B*) receiving requests from Sysbench [144] clients. The CPU usage percentage is collected over time as a consolidation metric while the transaction rate and the latencies are collected as performance metrics. The results are reported in Figure 2.2 and the remaining of this subsection details each of its rows, one at a time. There are five rows that correspond to five different cgroup configurations.

[26] Tasks that only display information on the screen or that collect user inputs.

During most of the experiment, both *A* and *B* receive a load of 250 requests per second, which consumes about 40% of CPU time, but there are three differences between the load given to *A* and *B*:

1. *B* has 8 concurrent client threads whereas *A* only has 2.
2. *A* has a lower load of 50 requests per second between times 50 and 70.
3. *B* has two extreme peak loads where it receives 1200 requests over a second. The first peak occurs at times 20 when *A* needs the CPU, and the second peak occurs at time 60 when *A* has a lower need.

The goal of this setup is to observe if isolation is enforced at time 20 and if consolidation is allowed at time 60.

Baseline (1^{st} row)

In the baseline configuration, *A* and *B* are only allowed to use one CPU core. They are executed alone and sequentially on the same machine to simulate physical hardware isolation and to avoid hardware differences. In other words, the trace collected from the execution of *B* is time-shifted to overlap the trace collected from the execution of *A*. As shown in Figure 2.2a, with only one CPU core, *B* is limited to 600 requests per second and is not able to handle its two instantaneous peak loads of 1200 requests. As a result, the requests of *B* are delayed and their response time increases (see Figure 2.2b).

This configuration shows the ideal performances of *A* and *B* at the cost of wasting more than 100% of the CPU time(see the idle percentage in Figure 2.2c).

Two CPU sets (2^{nd} row)

Before trying to consolidate idle CPU time, we first want to evaluate the performance isolation when running *A* and *B* together on the same machine at the same time. In this configuration, *A* and *B* are deployed on distinct cores with the cpuset cgroup, but the results show that the isolation is not as perfect as the theoretical baseline. Indeed, *A* and *B* have to share some CPU caches, access to memory banks and to the SSD.

Compared to the baseline results, Figure 2.2d shows a decent decrease in the transaction throughput of *B* during its first peak load. Moreover, there is a light overhead perceptible in terms of latency for both *A* and *B* (see Figure 2.2e). Overall, the total time spent on the CPU slightly increased (see Figure 2.2f).

Single CPU set (3^{rd} row)

This configuration is the first step towards consolidation. To save the idle CPU time (see Figure 2.2i), both servers are deployed in the same CPU cgroup which is allowed to run on a single core.

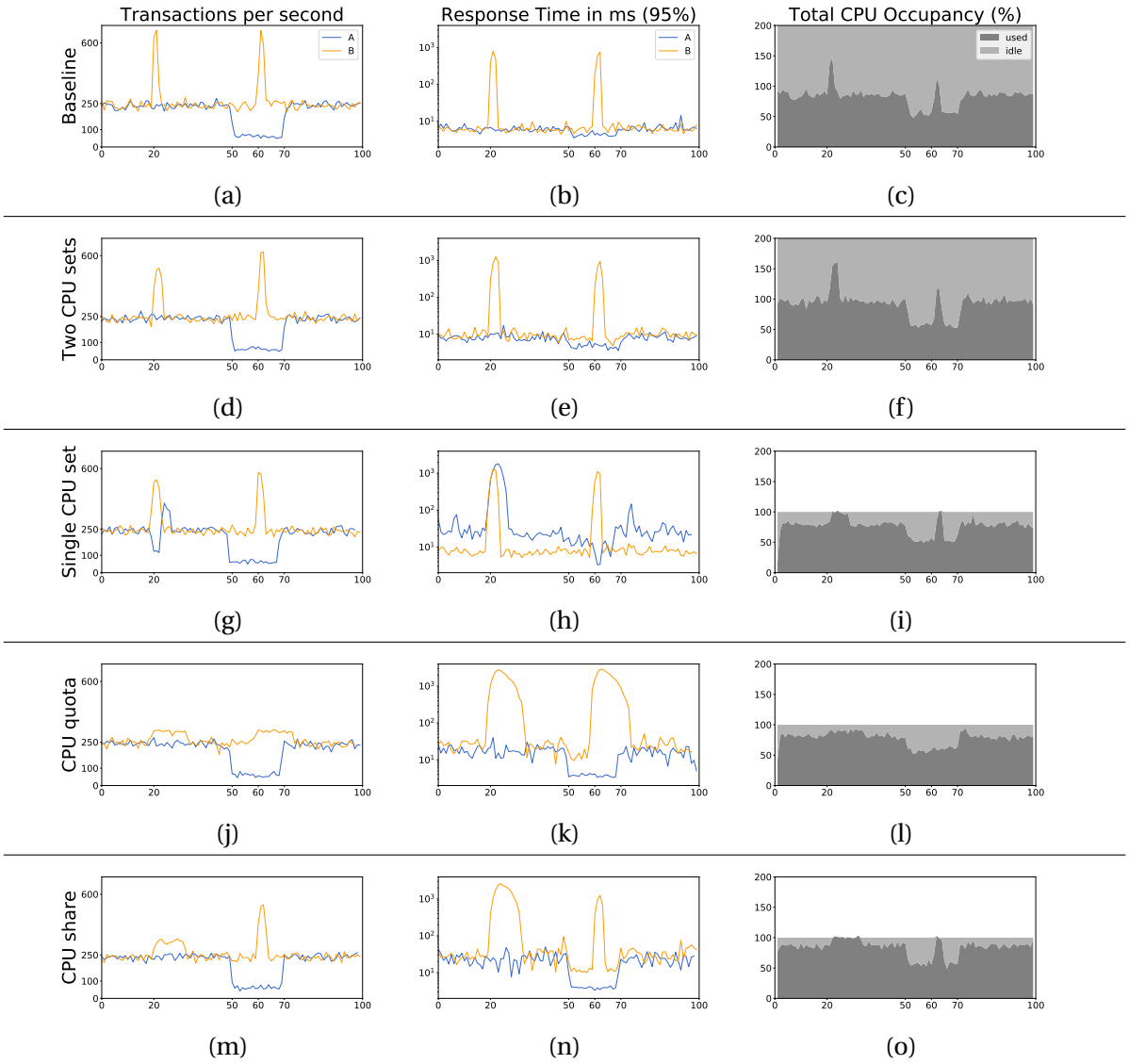


Figure 2.2: CPU Isolation and Consolidation.

Unfortunately, with this configuration, nothing prevents *B* from disturbing the performance of *A*. During its first peak load, *B* is able to lower the transaction rate of *A* (see Figure 2.2g). Consequently, the transactions of *A* are delayed (see Figure 2.2h), and *A* eventually catches up this delay by the time *B* stopped monopolizing the CPU. Moreover, we can notice that during the steady phases, the latency of *B* is better than that of *A*.

Despite its good consolidation, this configuration shows poor isolation.

CPU Quota (4^{th} row)

To tackle the lack of isolation observed in the previous configuration, the MySQL servers are deployed into separate cgroups configured with strict CPU quota bandwidth. When *B* is only allowed to consume half of the available CPU time, it cannot disturb the performance of *A*. Figure 2.2j shows that *A* has transaction rate similar to that of the baseline, but *B* can only increase its rate up to 300 per second to absorb the incoming 1200 transactions. Consequently, the period of time upon which its latency is degraded is extended from a few seconds to tens of seconds compared to the baseline (see Figure 2.2k). Moreover, during the steady phases, the latencies of *A* and *B* are fairly equal compared to the single cpuset configuration, but there still is an overhead compared to the two cpusets configuration.

Note that *A* has the same quota restriction as *B*, but in practice, it never needs to reach that limit. Unfortunately, *B* is not able to consume the idle time left over by *A*, especially when *A* experiences its lower load period (see Figure 2.2l). Therefore, despite its good isolation, this configuration is not ideal in terms of consolidation, as observed at time 60.

CPU Shares (5^{th} row)

To address the lack of consolidation observed in the previous configuration, the MySQL servers are deployed into separate cgroups configured with equal cpushares and without CPU quota bandwidth. As *A* consumes less CPU time than *B*, it is always prioritized over *B* when it is ready to execute.

Consequently, when cpushares are applied, the performance of *A* is still protected (see Figure 2.2m). But since *B* is allowed to overconsume the idle time left over by *A* (see Figure 2.2o), the period of time upon which its latency is degraded is shortened compared to the CPU quota configuration, as observed at time 60 (Figure 2.2n).

To summarize, these experiments suggest that “cpushares” is the most appropriate configuration to offer the best of both worlds: isolation and consolidation.

2.4.4 Block I/O, a time-based resource similar to CPU

As mentioned earlier, Input/Output accesses to block devices like SSDs are controlled with a scheduler. There are multiple I/O schedulers available in Linux, but the Completely Fair Queuing scheduler is currently the only one that can provide isolation and consolidation.

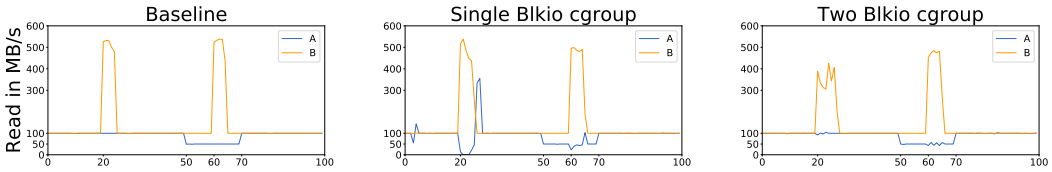


Figure 2.3: Block I/O Isolation and Consolidation.

To illustrate this property, we suggest the following experiment with two Filebench workloads (*A* and *B*) that bypass the page cache and sequentially read their own file of 1 GB on disk by blocks of 1 MB at a rate of 100 IOPS[27]. There are three differences between *A* and *B*:

1. *B* has 100 concurrent processes whereas *A* only has 2.

2. *A* has a slower pace of 50 IOPS per second between time 50 and 70.

3. *B* has two extreme peak paces where it attempts to do 500 IOPS over 3 seconds. The first peak occurs at time 20 when *A* needs bandwidth to the disk, and the second peak occurs at time 60 when *A* has a lower need.

The goal of this setup is to observe if isolation is enforced at time 20 and if consolidation is allowed at time 60.

Baseline (see Figure 2.3a)

In the baseline configuration, *A* and *B* are executed alone and sequentially to measure their throughput given a dedicated disk. Similarly to Section 2.4.3, the trace collected from the execution of *B* is time-shifted to overlap the trace collected from the execution of *A*. As shown in Figure 2.3a, it takes a few seconds to absorb the IO peaks of *B* because of the limited bandwidth capacity of the dedicated disk.

This configuration shows the ideal bandwidth isolation between *A* and *B*.

Single Blkio cgroup (see Figure 2.3b)

In this configuration, *A* and *B* are executed together inside the same cgroup and at the same time. As expected, the configuration offers no isolation because CFQ imposes fairness between processes. Thus, since *B* has more processes than *A*, it is able to disturb *A* whose bandwidth consumption drops to 0 MB/s at time 20.

[27] Input/Output operations per second.

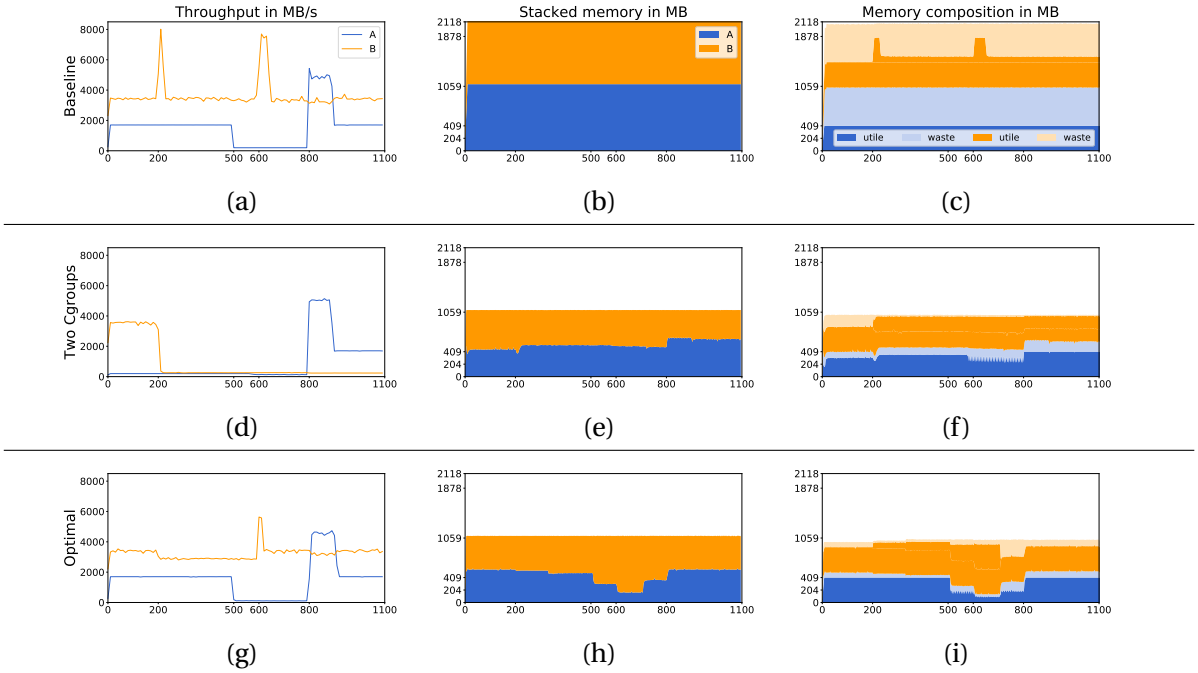


Figure 2.4: Memory is harder to isolate and consolidate.

Two Blkio cgroups (see Figure 2.3c)

In the final configuration, *A* and *B* are executed together in separate cgroups. We can observe that *B* does not disturb the bandwidth consumption of *A* at time 20, as it did in Figure 2.3b. Moreover thanks to consolidation, the second peak of *B*, at time 60, is shorter than its first peak at time 20.

Thus, this configuration offers both isolation and consolidation between *A* and *B*.

2.5 Memory, a spatial but not time-based resource

So far, we have seen that the consumption of time-based resource such as CPU and disk bandwidth can be easily controlled to provide isolation and consolidation. Even if memory could be considered as time-based resource, because of the bus bandwidth between the CPU and the RAM, it is mainly a spatial resource. In short, CPU and disk schedulers only have to decide which process should be using free resources. Unfortunately, there is no such thing as a memory scheduler, because the cost of systematically freeing memory is extremely high, especially compared to that of freeing CPU or disk bandwidth. Indeed, freeing CPU only requires a context switch and freeing disk bandwidth only requires a queue switch. On the other hand, freeing memory requires data to be swapped in and swapped out of memory to disk. Therefore, freeing memory is only done on demand, when a process attempts to access data that is not present in memory.

To illustrate that memory is harder to isolate and consolidate, we conducted the following

experiment with two Filebench workloads (*A* and *B*) that compete for memory to cache data. Both applications require at least 409 MB from a file called "utile", to process their load in time. In addition to their minimum memory requirement, the applications will waste memory by loading data from a file called "waste", that will never be reused. To measure the memory composition, we used the *mmap* and *mincore* syscalls to count how many pages of the files are in memory[28]. *A* and *B* have a few differences:

1. *A* does not need its memory between time 500 and 800.
2. *B* has two extreme peak where it needs 818 MB to process its load. The first peak occurs at time 200 when *A* needs its memory, and the second peak occurs at time 600 when *A* does not need its memory.

The goal of this setup is to show that memory is harder to isolate and consolidate.

Baseline (1^{st} row)

In the baseline configuration, *A* and *B* are allowed to consume up to 2118 MB of memory (see Figure 2.4b). We can observe on Figure 2.4a that *B* correctly responds to its two peak loads at time 200 and 600. Indeed, on Figure 2.4c, we can observe that *B* recycled some of its wasted memory to store useful data.

This configuration shows the ideal performances of *A* and *B* at the cost of wasting more than 1 GB of memory.

Two Cgroups (2^{nd} row)

In this configuration, *A* and *B* are not allowed to consume more than 1059 MB (see Figure 2.4e). This constraint is necessary to avoid the waste of memory (see Figure 2.4f). However, we can observe on Figure 2.4d that *A* does perform as expected at the beginning of the experiment, and that the performance of *B* is poor as soon as it hits its first peak load.

This configuration shows that Linux cannot isolate and consolidate memory.

Optimal (3^{rd} row)

In this configuration we used the *mmap* and *mlock* syscalls to manually schedule the memory placement and consumption. The schedule can be observed on Figure 2.4i:

- *A* locks its 409 MB of useful memory between time 0 and 500, unlocks it between time 500 and 800, and locks it again between time 800 and 1100.
- *B* locks its first 409 MB of useful memory during the whole experiment and only locks its second 409 MB between time 600 and 700.

[28] We reused the linux-ftools project [72].

Thanks to the schedule, *B* is able to process its peak load at time 600 without disturbing *A* during the whole experiment (see Figure 2.4g).

This configuration shows that 1059 MB is enough to run both *A* and *B*.

2.5.1 Conclusion

In contrast to VMs, containers are more resource efficient because they share the same kernel. Linux containers are built on top of key features such as namespaces, *seccomp* and cgroups. They able to consolidate CPU time and disk bandwidth while preserving performance isolation. However memory remains harder to consolidate without compromising isolation. The next chapter will explain why memory consolidation is still hard, and then Chapter 4 will demonstrate that even on the most simple scenario, Linux will fail to ensure isolation during consolidation.

CHAPTER 3

MEMORY AND CGROUP

This chapter introduces memory as a resource and explains how the Linux Kernel accounts for, limits and isolates memory through its cgroup feature. The related works presented at the end of this chapter concede that dynamically resizing the memory of cgroups is an unresolved problem.

3.1 Storing data in main memory

Main memory management is the topic of this thesis, but it is worth recalling that main memory is one level of the memory hierarchy. Main memory is spatially multiplexed through virtual addressing to enable concurrent usage and temporally multiplexed through virtual memory to extend its utilization.

3.1.1 Memory Hierarchy

Many components of a computer are dedicated to the storage of information. They are classified in the memory hierarchy based on response time and capacity. At the top of the hierarchy, registers provide the fastest access possible but they are very expensive and thus limited in size. At the bottom of the hierarchy, external memory—usually disks—can store data in massive quantities at an affordable price, but they are very slow and not directly accessible to the CPU. To mitigate that gap, each intermediate level is typically smaller and faster than the next level; programs have to take into account the data transfers—which are sometimes implicit—between levels.

Main memory is usually made of dynamic RAM (DRAM); a semiconductor storage that can handle read/write random accesses by storing each bit of data in a capacitor that requires

a periodic recharge. The CPU directly accesses main memory through an address bus and a data bus (hardware dedicated to communication).

The management of main memory is a critical duty undertaken by the kernel of the operating system. It allows programs to run concurrently without corrupting or stealing each other data. It provides ways to dynamically allocate, relocate and share portions of memory, and also to free it for reuse when it is no longer needed.

3.1.2 Spatial multiplexing

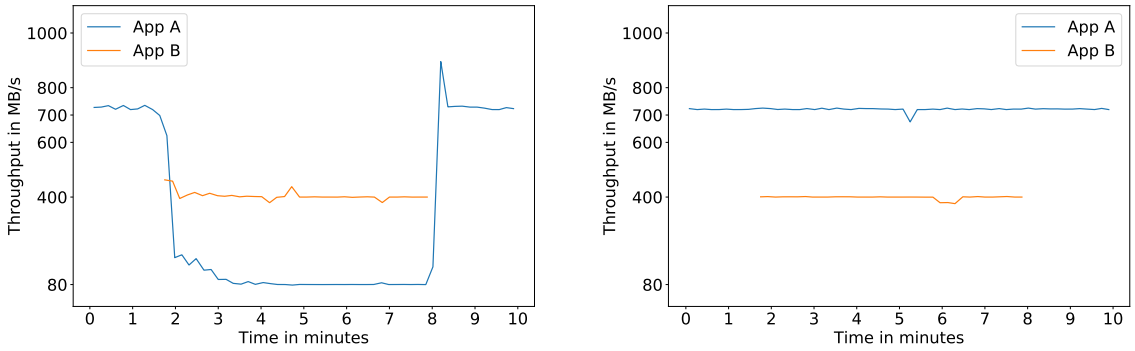
Thanks to the memory management, programs can have their own private view of memory called virtual address space. Indeed, it allows programs to use the same virtual address to store different objects because each program has its virtual address mapped to a different physical address. Virtual addressing is achieved through the segmentation unit and the paging unit of the memory management unit (MMU). The MMU is a hardware that translates virtual addresses into physical addresses. Segmentation is barely used in today's systems and most of address space virtualization is done through paging. The address space is partitioned into page frames (typically 4 KiB in length) and a set of page tables is introduced to specify how virtual addresses correspond to physical addresses. The page tables are stored in memory and are maintained by the kernel. When a new virtual address has to be translated, the MMU accesses the page tables to compute the corresponding physical address.

3.1.3 Temporal multiplexing

When a virtual address is translated into a physical address, a specific bit in the page table entry—a.k.a. the accessed bit—is set by the MMU. The accessed bit plays an important role in virtual memory: it notifies the kernel that the page is still in use and therefore should not be evicted out of memory.

Virtual memory is a feature of memory management that virtually extends the amount of available memory by performing implicit transfers of pages from memory to disk. As long as the set of pages immediately needed—a.k.a. the workingset [14, 123]—can fit in memory, the kernel can run programs whose total set of pages—a.k.a. the dataset—is larger than the available physical set of pages, by implicitly transferring pages in and out of memory as the workingset changes.

On the other hand, some programs have a very small memory footprint but tend to explicitly access data on disk. These programs can implicitly use more memory to store a copy of the data on disk in the hope that it will be re-accessed in the near future. Many applications, such as databases, rely on this caching technique provided by the kernel, to improve their performance.



(a) Without cgroups

(b) With cgroups

Figure 3.1: Illustrating the need for Memory Isolation.

3.1.4 The need for memory cgroup

In both cases, with virtual memory or disk access caching, isolation is required to control the amount of memory used to store the workingset or the cacheset. Indeed, it is very easy to accidentally collocate applications that interfere with one another. To illustrate these interferences, we suggest the following experiment with two Filebench workloads (*A* and *B*) that compete for memory to cache data. The application *B* often changes its small workingset, i.e., its set of most recently used data, but application *A* has a big static workingset. After running *A* for more than a minute, we start *B* and observe their performance. In the absence of cgroups, the dynamics of these workingsets cannot be detected and isolation cannot be guaranteed (see Figure 3.1). Indeed, when *A* is not isolated from *B*, *B* is able to flush the data of *A* out of memory. *A* only recovers its performance once *B* shutdowns at 8 minutes. But *B* does not really benefit from the extra memory stolen from *A*; indeed, its performance is the same in both cases.

In the absence of cgroups, the eviction policy of Linux can not detect the dynamics of these workingsets. We can deduce on the upper plot of Figure 3.1 that the data of *A* was evicted by the data of *B*.

3.2 Accounting and limiting memory with cgroup

As introduced in Chapter 2, cgroup stands for “control group”. It is a Linux Kernel feature that groups processes together to account for, limit and isolate their resource consumption. This section introduces the accounting and limiting feature of the memory cgroup[1].

[1] See `mm/memcontrol.c`.

3.2.1 Event, Stat and Page counters

Each memory cgroup has its set of counters that can be categorized into three types: Event, Stat and Page. Listing these counters gives an overview of the memory management concepts. Moreover, as we will introduce a new pair of event counters in Section 4.2.3, it is interesting to understand the differences with the current counters.

Event counters

An event counter counts the number of times something happened in a cgroup since its creation. It is a reliable counter to read since it is always incremented and never decremented; all modifications are recorded.

The reader might be familiar to the Linux `vmstat` counters which accounts for memory-related events at the scale of the whole machine[2]. Four of the same page events are provided per cgroup through the `sysfs`:

- `pgfault` and `pgmajfault` are exactly like the ones provided by `vmstat`. They count the total number of pages faulted by a cgroup. A page fault occurs when a virtual page has not yet been mapped to a physical page. The fault is said to be major if its content has been offloaded out of memory.

- `pgpgin` and `pgpgout` are different from the ones provided by `vmstat`. `pgpgin` (resp. `pgpgout`) counts the total number of pages charged (resp. uncharged) to a cgroup. `Vmstat` makes additional distinctions according to the origin of the page, `pswpin` and `pswpout` count transfers to and from the swap, `pgpgin` and `pgpgout` count transfers to and from the regular filesystem, and `pgalloc` and `pgfree` count allocations and freeing of pages.

The `/proc/vmstat` file also provides event counters related to the Page Frame Reclaiming Algorithm, but there are irrelevant to how memory is reclaimed with cgroup.

Dedicate limit events are available for cgroup:

- `low` counts the number of times a cgroup was shrunk despite being below its `min_limit`[3].

- `high` counts the number of times a cgroup was shrunk because it was over its `max_limit`[3].

- `max` counts the number of times a cgroup was shrunk because it hit its `hard_limit`.

- `oom` counts the number of times a cgroup had to trigger the out of memory function to kill a process to free memory.

[2] The Linux `vmstat` counters are exposed in the `/proc/vmstat` file.
[3] Exposed in version 2 of the cgroup interface.

Stat counters

Stat counters are less precise than event counters and should be considered as samples, because updates can cancel each other out and make monitoring tools believe that nothing has changed while increments and decrements are occurring at the same time. Nevertheless, they provide good hints on the workload needs in terms of memory.

- `cache` counts non-swap-backed pages and shmem/tmpfs pages.
- `rss` counts mapped swap backed pages and pages of the swap cache.
- `rss_huge` counts huge pages.
- `mapped_file` counts pages of files in memory accessible in the address space (no I/O syscalls).
- `dirty` counts pages not synced with their image on disk.
- `writeback` counts pages currently being synced with their image on disk.
- `swap` counts pages that should be in memory but are instead in the swap.

Additional stat counters also count the number of active and inactive pages per memory pools.

Page counters

Page counters are the backbone of memory isolation. They precisely count memory pages charged to a cgroup according to their type of use. Moreover, they raise exceptions when their limit has been reached[4].

- `mem`: counts pages that store user data/objects. It is the major page counter, the following counters are off topic.
- `kmem`: counts pages that store kernel data/objects.
- `tcpmem`: counts pages that store networking data/objects.
- `swap`: counts swap entries[5].
- `memsw`: counts the sum of `mem` and pages out of memory in the swap[6].

[4] `failcnt` counts the number of times the page counter reached its limit.
[5] All swap entries are taken into account even if there are clean copies in memory.
[6] swap entries of pages in memory are not taken into account.

3.2.2 `min`, `max`, `soft` and `hard` limits

A common misunderstanding about cgroup limit is that free pages are reserved or pre-allocated to a cgroup. Limits are just a set of policies which guides the memory reclaims. The major limitation feature is the `hard_limit`.

`hard_limit`

By default, when a cgroup is created, its resource consumptions are unlimited. The `hard_limits` of its page counters are initially set to $+\infty$[7]. Setting the limit of a cgroup does not reserve nor pre-allocate free pages, it only sets the `hard_limit` value of the page counter[8]. Free pages are managed by a binary buddy allocator and they do not belong to any cgroup[9]. As soon as they are allocated and contain valuable data, they are charged to the cgroup that issued the request.

As the `hard_limit` must never be breached, the cgroups are first shrunk, if needed, before charging a new page. During this process, the cgroup may be led to kill some of its tasks to free memory because it is Out Of Memory (OOM).

Since cgroups can be nested to form a hierarchical tree, their page counters are also nested. When a page is charged to a leaf cgroup, the internal cgroup nodes on the path to root cgroup are also charged[10]. Multiple cgroups can be shrunk at once to respect the `hard_limit` of an internal node[11]. Memory pool shrinking is discussed in Section 3.3.1.

`min`, `max` and `soft_limit`

Three additional limits have been added for fine-tuning:

The `min_limit` feature gives some protection to the cgroups whose current usage is below their minimum threshold. Memory is first reclaimed in cgroups whose current usage is above their minimum threshold, but if memory is still scarce, this protection is not guaranteed. By default, the `min_limit` is initialized to zero.

The `max_limit` feature is not as strict as the `hard_limit`. Cgroups can exceed their `max_limit` without triggering the OOM killer. But when they do exceed it, they are immediately and periodically forced to run additional attempts to shrink their memory pools. This extra work ensures that the data of the cgroup are still worth keeping in memory despite the `max_limit` being exceeded because the data are still being accessed. But the `max_limit` is not a silver bullet because its periodic memory reclaims consume extra CPU time and most importantly, it does not work with applications that store their data in the type of pages which are easy to reclaim. Indeed, these kinds of applications might never be able to exceed their `max_limit` if they have to.

[7] The maximum value of an unsigned long integer.
[8] Setting the `hard_limit` too low may fail if the size of the cgroup cannot be shrunk.
[9] See `mm/page_alloc.c` file.
[10] If the `use_hierarchy` parameter is false, the hierarchy is flat under the root cgroup.
[11] The root cgroup is always unlimited.

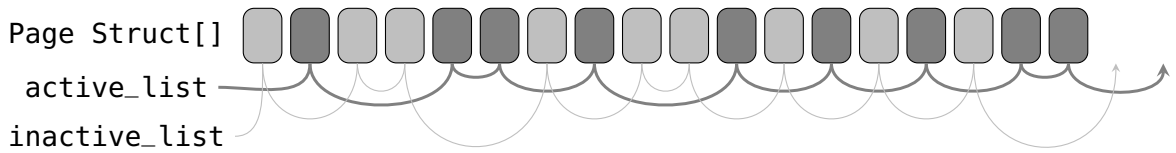


Figure 3.2: Linux Memory Pool: `lru`.

The `soft_limit` feature allows system administrators to define a desired or preferred cgroup size. If memory is available, the cgroup can grow beyond the `soft_limit` without breaching the `hard_limit`. But as soon as memory gets scarce, the cgroup whose `soft_limit` is the most exceeded will be shrunk in priority. By default, the `max_limit` and the `soft_limit` are initialized to $+\infty$.

3.3 Isolating cgroup memory reclaims

Cgroup isolation goes beyond the quantitative counters and limits presented in the previous section. Cgroup also provides a qualitative isolation because each cgroup has its own set of memory pools. This section is important because it introduces memory tracking technics used in Chapter 6 and explains why we cannot revert to a centralized pool of memory.

3.3.1 Linux memory pool

The goal of a memory pool is to measure the utility of the pages it contains. When memory is running low, it has to guess the page whose next use will occur the farthest in the future, and evict the page out of memory to free space [7]. The algorithm is known as the Page Frame Reclaiming Algorithm or Page Frame Replacement Algorithm (PFRA)[12]. The PFRA of the Linux kernel is quite unique but does borrow some idea from the literature: like 2Q [25] it uses two queues, like LIRS [23] it uses a refault distance, and like Clock-Pro [22] it keeps in memory information about recently evicted pages.

The memory pool of Linux is called the `lru`, but it does **not** strictly follow the Least Recently Used (LRU) order. The `lru` is composed of two double linked lists of pages called the `active_list` and the `inactive_list` (see Figure 3.2). Both lists are mostly manipulated to respect the First In First Out (FIFO) order. The former list tends to include pages that have been accessed recently and therefore require more protection. The latter list tends to include pages that have not been accessed for some time and therefore are good candidates for eviction. When the `inactive_list` is big enough, the `active_list` is never shrunk. This rule protects active pages from access patterns such as scanning through a whole database. But when the size of the `inactive_list` is detected as low during the PFRA, some pages from the `active_list` are moved to refill the `inactive_list`.

[12] See `mm/vmscan.c`.

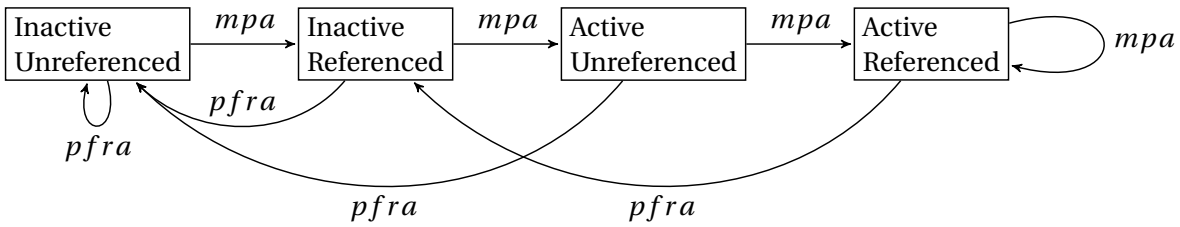


Figure 3.3: Page state automata in Linux memory pool.

Newly allocated pages can be inserted in either list according to heuristics. For example, a page from a file that was recently seen in memory will be inserted in the `active_list`. On the other hand, since page faults severely delay programs, the page is placed in the `active_list` to avoid other faults on the same page in the near future. In Section 6.3.2, we used these page demand events to track the activity of cgroups.

As monitoring memory accesses is very expensive, the page activity is mainly tracked through two light mechanisms. These mechanisms are almost the same as the ones used by the idle page tracking tool in userspace that we are going to cover in Section 5.2. The monitoring is mostly focused on pages in the `inactive_list` because they need to be promoted to the `active_list` as soon as possible for protection. On the other hand, it is pointless to monitor active pages since they are already protected and the likelihood of seeing another access on them is pretty high.

The first monitoring mechanism is directly catchable by the kernel: When a page is accessed in kernel mode—for e.g., during a read syscall on a file—a specific kernel function[13] is called to remember that the page was accessed. In addition to the bit `PG_active` used to remember on which list the page is stored, a second bit, the bit `PG_referenced`, is used to remember that the page was recently referenced. The PFRA also manipulates these bits and the automata in Figure 3.3 provides a glance at how pages transition from one list to the other. In Section 6.3.2, we used the page activation events, i.e., when pages are inserted in the `active_list`, to track the activity of cgroups.

The second monitoring mechanism used to track memory accesses is the bit accessed of the page table entry. As mentioned before, this bit is set by the MMU to notify the kernel that the page was accessed in user mode. But the kernel consults and resets this bit only when it has to shrink the `lru`, because it is expensive to reverse map a physical address to all its virtual addresses.

3.3.2 Splitting memory pools

Memory pools are extensively used in the kernel. There are memory pools for each NUMA node, for each address zone, for each cgroup and for each usage type of page. By splitting memory pools, Linux improves its ability to manage memory. For instance, NUMA nodes

[13] `mark_page_accessed()` is shortened as `mpa` in Figure 3.3.

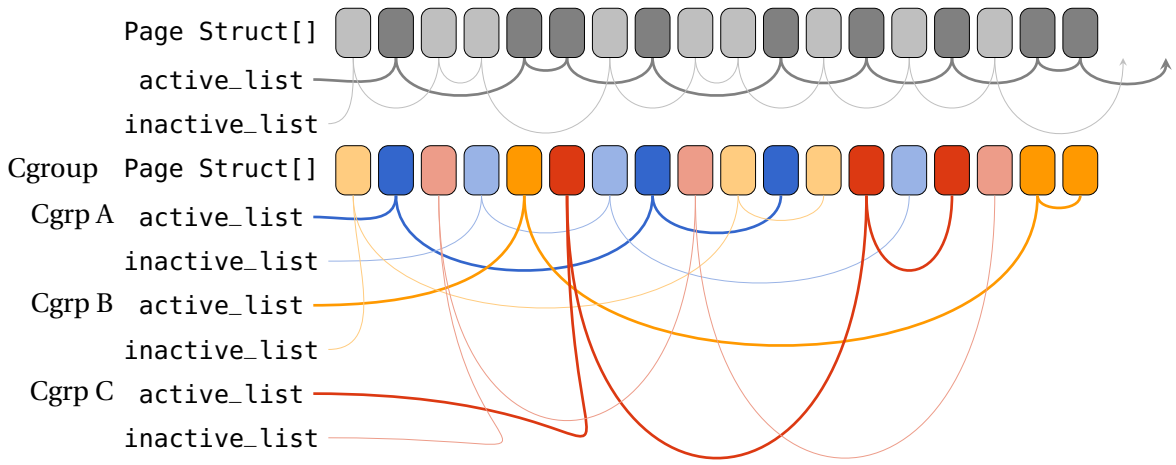


Figure 3.4: Duplicated local cgroup lists were "bolted" on the global ones.

and address zones are statically sized physical constraints that need to be balanced from a hardware point of view. On the other hand, the cgroup and usage dimensions are not bounded by hardware characteristics and are therefore extremely dynamic.

Page type `lru` splitting

Pages can store any kind of data, they are tagged according to their type of use. We don't have to discuss every usage types here, but it is worth mentioning that Linux manages each type slightly differently during the memory reclaim. For example, `unevictable` pages[14] store data pinned in memory by users and `slab` pages store kernel objects. But this thesis will only focus on the following two types: pages that store persistent data and those that store non-persistent data. The former are called `file` pages[15] because they have an image in a file on disk, and the latter are called `anon` pages[16] because they are anonymous, i.e., they do not have a dedicated location on disk. When `file` pages are evicted out of memory, their data are synced back on disk through a regular filesystem. When `anon` pages are evicted out of memory their data are backed up in an area called the swap.

In 2008, R. van Riel introduced the `anon` and `file` `lrus` [151]. Linux stores `file` and `anon` pages in separate typed pools because it needs to (*i*) measure and compare the utility of a specific type of page, (*ii*) efficiently reclaim pages of a specific type and (*iii*) take advantage of the different transfer rates of the swap device and the file device. If both types used the same pool, the PFRA could waste time on filtering pages according to their type or could move them around and disturb their recency order.

[14] `Unevictable` pages have their `PG_unevictable` flag set.
[15] `File` pages have their `PG_swapbacked` flag cleared.
[16] `Anon` pages have their `PG_swapbacked` flag set.

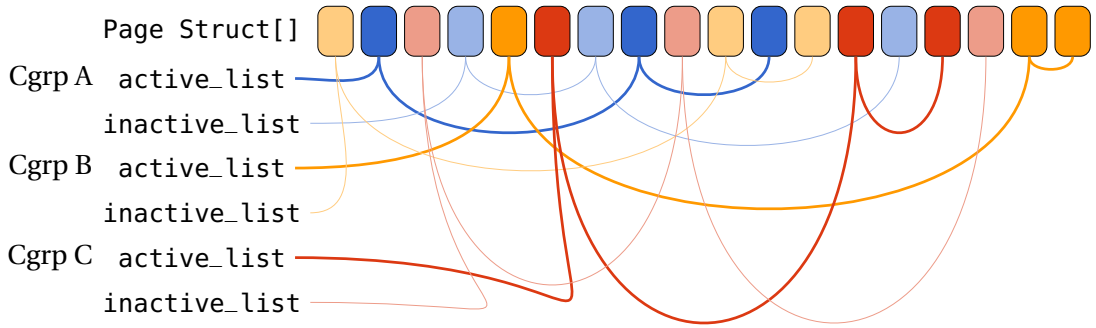


Figure 3.5: J. Weiner removed the global set of lists and split it into local sets of lists per cgroups.

Cgroup `lru` splitting

Before J. Weiner's work of 2011 [153], cgroups were implemented with local lists "bolted" on the global ones (see Figure 3.4). These additional lists allowed the kernel to quickly iterate through pages belonging to a specific cgroup by skipping pages from other cgroups, but J. Corbet reported that this design had at least three disadvantages [91]. First, the global reclaim did not take into account the cgroup limit policy. Second, to keep duplicated structures synchronized, only one cgroup could be reclaimed at a time, but this created interference between cgroups. Third, duplicated structures not only added complexity in the code but also increased the memory footprint. As a result, J. Weiner removed the global set of lists and split it into local sets of lists per cgroups (see Figure 3.5).

3.4 Resizing dynamic memory pools

Unfortunately, splitting memory pools comes with the unresolved problem of dynamically resizing the pools. A resizing heuristic for the `anon` and `file` `lru` was introduced at their birth at later refined. But resizing `lrus` of different cgroups is harder because they must remain isolated from each other.

3.4.1 Resizing `anon` and `file` memory pools

Resizing dynamic pools is hard, but Linux has come up with three cases to balance the ratio of `anon` and `file` pages according to their utility. When `file` pages are more accessed than `anon` pages, Linux will prefer to reclaim `anon` pages and vice versa. This property is later illustrated in Section 5.1.2. Moreover, Linux also adjusts the intensity at which pages are reclaimed. Indeed, as killing a process is worse than slowing all the processes, the PFRA will do its best to avoid triggering the Out Of Memory killer (OOM). On the other hand, every time a page has to be reclaimed, the PFRA should not hog the CPU for too long.

The amount of work done by the PFRA is tailored to fit the current situation, whether to answer the balance need between anon and file pages, or to answer the balance need between the throughput and the latency of the PFRA's execution. This logic is described in the get_scan_count function: it computes W_X, the amount of work done by the PFRA, i.e., the number of pages to scan in the lru X ($X \in \{\texttt{file}, \texttt{anon}\}$). The function get_scan_count has three different strategies which are "shrink both equally", "shrink one only" and "balanced shrinking".

Shrink both equally

The PFRA proceeds by waves with increasing intensity. If there are N_X pages in the lru X, the first wave will work on $\frac{N_X}{2^p}$ pages, where p is the scan priority and is set to 12 at the first wave. If the first wave fails to reclaim enough pages, the PFRA will try again but the next wave will work on twice as many pages, i.e., p is decremented after each wave. On the final wave, where p is set to 0, the PFRA will not try to apply any balancing cleverness between the anon and file lrus. Both lrus will be fully scanned. Therefore, in this case, $W_X = N_X$.

Shrink one only

Obviously, if swapping is not allowed, the anon lru will not be shrunk. There are many cases where this can occur: when the page counter memsw has reached its limit, when the swap device is full, or when the user set swapiness to zero.

When swapping is allowed, programs usually expect that anon pages are more likely to be in memory than file pages. Therefore, the PFRA will choose to shrink only the file lru if both of the following conditions are met on $N_{\texttt{inactive_file}}$, the size of its inactive_list:

- $N_{\texttt{inactive_file}} >= N_{\texttt{active_file}}$
- $\frac{N_{\texttt{inactive_file}}}{2^p} > 0$, i.e., not during the first waves unless $N_{\texttt{inactive_file}} > 2^{12}$.

In the two aforementioned cases, $W_{\texttt{anon}} = 0$ and $W_{\texttt{file}} = \frac{N_{\texttt{file}}}{2^p}$.

The PFRA will almost never choose to shrink the anon lru only. There is a very rare case, called "cache trap", where a "feedback loop" will tend to wrongly prefer to evict file pages. In that case, a heuristic will try to detect this "feedback loop" and ask to shrink the anon lru only ($W_{\texttt{file}} = 0$ and $W_{\texttt{anon}} = \frac{N_{\texttt{anon}}}{2^p}$).

Balanced shrinking

In 2008, R. van Riel also introduced a method to choose between the anon and file lrus [151]. The balanced shrinking is guided by a user-defined parameter called swapiness and by two counters per lru called recent_scanned and recent_rotated. The swapiness is a positive integer smaller than 100 and set to 60 by default. The recent_scanned counts the number of pages recently scanned, i.e., the pages on which the PFRA has been working on.

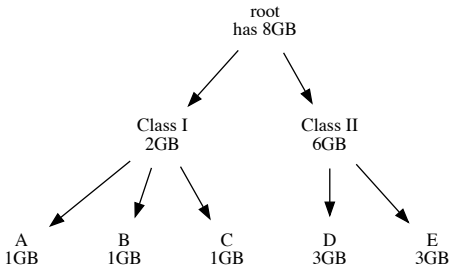


Figure 3.6: Example of cgroup hierarchy with `hard_limit` configuration.

The `recent_rotated` counts the number of pages recently scanned but not reclaimed and kept in memory because they were recently accessed. The counters are called "`recent`" because they are halved periodically when `scanned` reaches the quarter of the size of the `lru`.

The counters are used to measure the effectiveness of memory reclaims. We decided to refer to this metric as the "*rotate ratio*". It is computed for the `anon` and the `file` `lrus` as follows:

$$R_X = \frac{\texttt{recent_scanned}_X}{\texttt{recent_rotated}_X} \quad X \in \{\texttt{anon}, \texttt{file}\} \tag{3.1}$$

Then, to obtain the values of W_X, the following formula is applied:

$$W_X = \frac{N_X}{2^p} \times \frac{Q_X \times R_X}{Q_{\texttt{anon}} \times R_{\texttt{anon}} + Q_{\texttt{file}} \times R_{\texttt{file}}} \quad X \in \{\texttt{anon}, \texttt{file}\} \tag{3.2}$$

where Q_X[17] is the `swapiness`.

3.4.2 Resizing cgroup memory pools

Resizing cgroup memory pools is still an unresolved problem. Suppose a machine has 8 GB of memory; it can divide it between two classes of application. The first class is applications that can overcommit and share their memory because they do not need it at the same time (A, B and C). But the second class are high priority applications that cannot share resources (E and D). A cgroup hierarchy configuration corresponding to that example is given in Figure 3.6. Which cgroup—between A, B and C—should be shrunk when the parent reaches its 2 GB limit? In 2002 Michael Kerrisk [130] said that "there is no real agreement on the semantics of how the hierarchy should be walked and pages reclaimed". The current im-

[17] $Q_{\texttt{anon}} = \texttt{swapiness}$, and $Q_{\texttt{file}} = 200 - \texttt{swapiness}$

plementation has no specification and will try to reclaim memory from any of them proportionally to their current size.

The formulas 3.1 and 3.2 could be generalized for multiple cgroups (see formulas 3.3 and 3.4 where R_X is the *rotate ratio* of the cgroup X and Q_X is a new user-defined priority value). But W_X is a number of pages to scan, not to reclaim. This method would, therefore, require all cgroups to be scanned whenever one cgroup reaches its local limit. Unlike the `anon` and `file` `lrus` of the same cgroup, the `lrus` of different cgroups cannot be synchronously scanned because it would break the isolation between the cgroups. Nevertheless, in Section 6.3.1, we use the *rotate ratio* in one of our approach, defined as follows:

$$R_X = \frac{\texttt{recent_scanned}^X_{\texttt{anon}} + \texttt{recent_scanned}^X_{\texttt{file}}}{\texttt{recent_rotated}^X_{\texttt{anon}} + \texttt{recent_rotated}^X_{\texttt{file}}} \quad X \in \{\texttt{A}, \texttt{B}, \texttt{C}...\} \tag{3.3}$$

$$W_X = \frac{N_X}{2^p} \times \frac{Q_X \times R_X}{\sum Q_i \times R_i} \quad X \in \{\texttt{A}, \texttt{B}, \texttt{C}...\} \tag{3.4}$$

In 2016, V. Davydov [101] highlighted the same problem. When all containers are actively demanding pages, the greediest container whose demand rate is the highest can outrun the others, and hog most of the memory. He stated that this behavior was unfair, especially in the case where the former container reclaims useful pages from others in favor of useless pages, i.e., pages that are used once and never used again.

We believe that greedy containers should not be systematically throttled by the kernel. On the contrary, greed is good because it enables the demand-as-you-go resource model. Unless there were a way to predict that the demanded pages are going to be useless, greed should prevail.

Five ideas were suggested during V. Davydov's talk to address this issue: a "dedicated system daemon", a "timestamp on each page", the "refault distance", the "vmpressure" and "memdelay".

Dedicated System Daemon

The `min`, `max` and `soft` limits were some first steps to solve the problem at a low cost, but users generally don't know how to set these limits and even the `hard` limit is not obvious to determine [52]. Besides, the `min` and `max` are absolute limits which do not compare cgroups together. The `soft` limit policy selects the cgroup whose limit is the most exceeded, but this cgroup could be the one the most in need.

J. Weiner recalled that `soft` and `hard` limits can and should be dynamically adjusted to avoid the problem by "manually" routing memory pressure from userspace decisions [101]. A system daemon would measure how much memory pressure is created by each cgroup and make its limiting decisions accordingly. But it is not clear how that pressure would be detected and quantified. In Section 6.3.1, we present a similar metric-driven approach that takes decisions in userspace; but our implementations never modify the `hard_limits`

because we want the transfers to occur at the very last moment, i.e., when a growing cgroup asks for more pages.

Timestamp each page

V. Davydov suggested tracking the oldest page on each list by storing the time at which **each** page was added to the `lru` [101]. The proposed solution could then try to achieve an approximate balance of ages. As J. Weiner abolished the global LRU order [153] in the interest of isolation, the best approximation can only be achieved by reconstructing a global FIFO order. But since FIFO policy has too many flaws compared to LRU, such as Belady's anomaly [8][18], the global order would be restricted to the `active_list` only. As no implementation of this solution is available, we do not know if the balance of ages can preserve the isolation between cgroups. For instance, a small cgroup can quickly churn its active pages in and out of the `active_list` and indirectly force the balancing mechanism to shrink the `active_list` of other cgroups. Besides, the most reluctant aspect of this approach is its memory cost: It simply duplicates information because timestamps and lists both encode the same ordering information. While the former is absolute, the latter is relative. Nevertheless, in Section 6.3.2, we show that storing a **single** timestamp per cgroup is enough to improve the current strategy of Linux.

Refault Distance

In 2012, J. Weiner introduced the refault distance which measures how long evicted pages stay out of memory before being faulted back in [92, 154]. In short, a clock is incremented every time a page is removed from the `inactive_list`, either to evict it or to activate it. When a page is evicted, the kernel recycles a "pointer to that page" in memory to store the age of its eviction. Afterwards, when the page is faulted back in, the kernel compares the age of its eviction to the current value of the clock. This difference, called the "refault distance", is used to detect if a page was recently seen in memory. A heuristic decides that the page has to be promoted to the `active_list` if its refault distance is smaller than the current size of the `active_list` because it assumes that the page should not have been evicted. Otherwise, if the refault distance is greater than the size of the `active_list`, the page is simply placed in the `inactive_list`. This metric was suggested to solve the resizing of cgroup because it can be used to know when to grow a cgroup. Unfortunately, it cannot be used to know when to shrink a cgroup. Another inconvenient of the refault distance is that it does not work on the `anon lru` because it requires to recycle an unused field in a data structure that is not available for `anon` pages.

vmpressure

In 2012, A. Vorontsov introduced vmpressure notifications to userspace [152, 93]. Four types of events are sent to applications that wish to voluntarily reduce their memory footprint when virtual memory is under pressure.

[18] Belady discovered that an increase of memory can result in an increase of page faults with FIFO.

- `VMPRESSURE_LOW`: The system is out of free memory and has to reclaim pages to satisfy new allocations. However, there is no particular trouble in performing that reclamation, so the memory pressure, while non-zero, is low.
- `VMPRESSURE_MEDIUM`: A medium level of memory pressure is being experienced; enough, perhaps, to cause some swapping to occur.
- `VMPRESSURE_CRITICAL`: Almost no page in memory is a suitable candidate for eviction.
- `VMPRESSURE_OOM`: Memory pressure is at desperate levels, and the system may be about to fall prey to the depredations of the out-of-memory killer.

The averaged vmpressure in a cgroup is computed over a window periodically, when the total number of pages scanned in both `lrus` of a cgroup reaches a threshold (see formula 3.5). The `LOW` event is always triggered at the end of the window, the `MEDIUM` event is triggered if the vmpressure is above 60, and the `CRITICAL` event is triggered if the vmpressure is above 95. On the other hand, the `OOM` event is not related to vmpressure formula, but instead, it is triggered when the scan priority drops below 3.

$$vmpressure = 100 \times \left(1 - \frac{reclaimed}{scanned}\right) \tag{3.5}$$

In essence, the vmpressure measures how hard/easy it is to reclaim memory which does not really reflect the need for memory. Moreover, according to peers, it does not look like a reliable metric. M. Hocko said that vmpressure only works well on small systems but on larger systems, pressure tends to look high even when the situation is not that severe [101]. We believe that he experienced false positive notifications because the measurement window is not suited for large systems[19]. Adjusting the window at runtime could resolve this issue. Surprisingly, J. Weiner experienced the exact opposite problem [155]. On machines where memory is in the hundreds of gigabytes and SSDs reduce the speed gap between memory and disks, the rate at which memory can be reclaimed is as fast as the rate at which memory can be scanned. In this condition, notifications are triggered at the very last moment, just before the OOM event. We believe that the real ratio value should also be exposed to monitoring applications since the 60 and 95 thresholds may be obsolete.

memdelay

In 2017, while we were working on this problem, J. Weiner submitted a patch that measures the time spent "waiting for memory", i.e., memdelay [155, 156]. He observed that the execution of the PFRA can overwhelm an already overloaded CPU. As mentioned before, when the speed gap between memory and disks is reduced, the bottleneck of memory-intensive applications can shift towards CPU time. Historically, these workloads would tend to block

[19] The `vmpressure_win` is statically set to 512 pages.

on IO completion, but they would be easily identified because of their idle CPU time. Today, the paradox is that by increasing memory capacity, one can free up unproductive CPU time spent in the PFRA to boost application throughput and latency. J. Weiner aims at resizing cgroups according to the measured memdelay. If the delay is above a threshold, the cgroup should be grown and if the delay is below another threshold, the cgroup should be shrunk. According to J. Weiner, the memdelay is more user-friendly than other memory pressure metrics because it does not rely on hardware aspects to be meaningful. For instance, if 2000 pages are refaulting per seconds from an SSD, the situation is not as bad as if they were refaulting from a rotating disk.

Unfortunately, P. Zijlstra was skeptical about this first prototype since *i)* it can slow down critical paths in the scheduler code, and *ii)* it does not reuse some of the existing scheduler counters. In some cases, the execution of the PFRA could simply be sped up by further dividing the big cgroups into smaller cgroups to shorten the length of the page lists.

3.5 Conclusion

We have introduced memory as a resource managed in units of pages. The utility of each page is tracked by a special data structure called `lru`. Kernel developers have weighed the pros and cons of partitioning the global `lru` into multiple `lrus`. In the process, they introduced the problem of dynamically resizing the `lrus`. A heuristic is provided to solve this problem for the `lrus` of the same memory cgroup, but there is no real agreement on how the `lrus` of different cgroups should be resized. In the next chapter, we will present a case where there is no ambiguity about which cgroup should be shrunk.

CHAPTER 4

ISOLATION FLAWS AT CONSOLIDATION

In the previous chapter, we introduced the general problem of dynamically resizing the memory pools of cgroups. In this chapter, we are going to narrow down the problem and restrain it to our activity model and hypothesis. We claim that the performance of active containers should not be disturbed if there is unused memory in inactive containers. We demonstrate that during consolidation, Linux cannot sustain the performance isolation of the most active containers, and thus, even in the simplest of cases, i.e., when a single container is obviously the sole active container. Our first contribution highlights that the need for isolation has incapacitated the Linux kernel to consolidate a class of memory activity pattern that used to be easy to consolidate without isolation. **The goal of this thesis is to preserve isolation during consolidation.** Afterwards, in Chapter 7, we will demonstrate that our solutions reach this goal on less obvious cases where there is still some activity remaining in the inactive containers.

4.1 Modeling Consolidation

As mentioned before, consolidation is about taking advantage of temporal multiplexing opportunities. But mitigation mechanisms are required to handle all the possible behaviors of real-life applications. As simple as it might be, our model is based on a real case study, business-inspired by Magency's applications[1].

4.1.1 Model assumptions

We will not discuss finding the temporal multiplexing opportunities in real workloads, as it is off topic. On the one hand, this information can be finely detailed so that the memory

[1] We recall that this thesis has been sponsored by Magency.

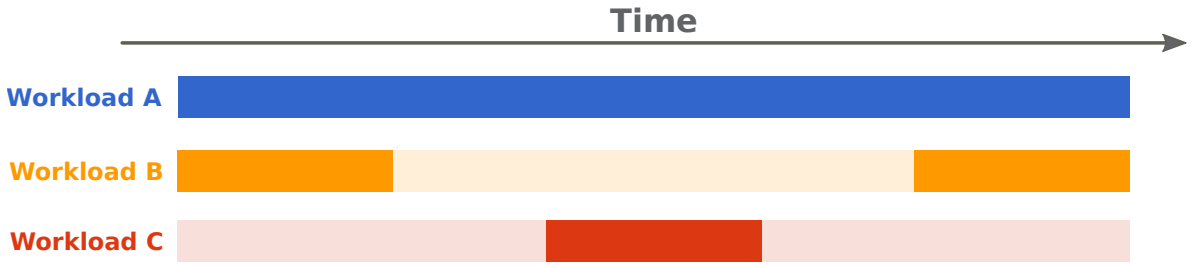


Figure 4.1: An example of multiplexing opportunity.

requirements of each workload are known at any time. On the other hand, the information can be almost vague so that the aggregated memory requirements of all the workloads are known to be always lower than the sum of their `hard_limit`. As the latter assumptions are easier to provide than the former at the level of the cluster orchestrator, we decided to make the following model assumptions:

1. In our model, applications are classified into two classes. They are either **active**, meaning that they need all the memory they asked for to ensure the isolation of their performance, or **inactive**, meaning that with less memory, they could perform as well as they are performing with their current consumption.

2. At any time, applications can switch from one state to the other, but the orchestrator does not have to guess the exact moments.

3. The aggregate memory needs of active users do not exceed the physical capacity, i.e., when an application becomes active there is always another application that recently became inactive (provided that the applications have the same memory footprint, but this one-to-one ratio is only for convenience. In practice, any ratio could hold).

Figure 4.1 depicts a consolidation scenario example, where three applications A, B, and C are hosted on the same machine. A is always active but B and C are never active simultaneously. As B deactivates, its memory becomes useless. But when C activates in the middle of the scenario, we would like the Linux kernel to automatically transfer memory from B to C to avoid waste. Afterwards, when B activates again, we would like to transfer back the memory of C to B. This scenario is also very likely to occur on a personal workstation: A could be an application live streaming your screen, B could be your IDE that builds your application and C could be a virtual machine that tests your application.

Obviously, if the scenario was known in advance, the user could schedule the memory transfers before the activations by using specific system calls such as `madvise`, `fadvise` or `mlock`. Another way of scheduling memory transfers beforehand is to use cgroup `soft_limit`. When a cgroup deactivates, the orchestrator can set its `soft_limit` to the minimum (for

e.g., 0) and when it activates, its `soft_limit` can be set to the maximum, i.e., to the `hard_limit`; but as we will see in Section 6.2, `soft_limits` are not ideal for the task.

Out of the lab, these consolidation events should be considered as random and unpredictable. Instead of trying to avoid consolidation by adjusting `hard_limits`, the goal of the thesis is to provide mechanisms to plan the reaction of the kernel in the event of consolidation. The decisions are based on activity predictions and can be taken from userspace or kernelspace. On the one hand, userspace predictions are built on metrics, see Section 6.3.1, but the decision tends to lag because they are not close to the memory management mechanisms and events available in the kernel (recall Section 3.3.1). On the other hand, kernelspace predictions are built on kernel events, see Section 6.3.2, but the decision must remain agnostic about application feedback. As the current strategy of Linux does not take into account the activity of cgroups, we will see in Section 4.2.1 that the kernel does not preserve the isolation of the most active containers.

4.1.2 Countermeasures

In the previous subsection, we introduced assumptions that does not always hold for every applications. In this subsection, we suggest countermeasures to mitigate these corner cases:

1. The memory activity of some applications might be impossible to track or compare with generic methods. These applications will therefore be harder to automatically consolidate. We recommend to treat them separately by falling back to methods that dynamically readjust their `hard_limit`.

2. There is a limit to how fast applications can switch from one state to the other. The minimum context switch time is mainly constrained by the amount of data used by the application and by how fast these data can be reloaded from disk. Increasing the disk bandwidth can provide quicker context switch time, but at some point, it might be better to consider the application as being active over the period of time where its inactivity time is less than twice the context switch time.

3. If all applications are active and their aggregate memory needs exceeds the physical capacity, then three solutions are available: (*i*) they could all be thrashing, (*ii*) by defining static or dynamic priorities, only the least important ones could be thrashing, and (*iii*) some applications could be migrated to a newly booted machine.

We did not explore these overcommitted scenarios, as designing mitigation mechanisms was not our goal. Our aim was to study consolidation when there should be enough memory, i.e., when there are enough inactive containers. The case where all containers are active is discussed in Chapter 8.

4.1.3 Industrial Application at Magency

The activity model exposed in the previous subsection was inspired by a business pattern observed at Magency [73]. This company sells collaborative applications which target meetings, trainings, and corporate events. The workload in this environment is very heterogeneous: during an event, the application is always active, but before and after the event, the application often changes its activity status because it is sporadically accessed to upload or download content. Magency wanted to use containers as a means to consolidate the resources of applications when they often change their activity. But during events, the applications encountered momentary slowdowns because they were running out of memory. Yet much of the memory was unused because it was allocated to other inactive applications.

Their first fix was to boot containers on demand. A front-end container receives all the incoming traffic and redirects the requests to individual containers according to the client's identity. If no requests were seen for an amount of time on a specific container, the front-end container would simply shut it down. Upon receiving a request routed to a container stopped, the front-end container would delay the request while booting its container. Unfortunately, despite saving most of the resources, this solution had major disadvantages:

- The front-end container became a scaling bottleneck.
- As containers were stateful applications, their boot time was not negligible.
- As the front-end container could control the life-cycle of other containers, it became a security vulnerability that increased the attack surface.

Magency, therefore, decided to invest in research to improve the memory consolidation of the Linux kernel.

4.2 Consolidation: once a solution, now a problem

Consolidation used to be a solution but it has now become a problem[2]. Indeed, booting a container to reuse the memory of sleeping containers can degrade the QoS of running containers. Today, users first have to manually free the memory of sleeping containers before booting a new one; but in the past, this simple scenario used to be consolidated without errors when containers were not required.

4.2.1 Consolidation with containers

We have reproduced in the laboratory the consolidation problem encountered at Magency with real applications that are MySQL [75] (App *A* and *B*) and Cassandra [58] (App *C*) by using the following scenario. *A*, *B*, and *C* are deployed on a physical machine with enough

[2] A smart man once said: "In life, you would rather be the problem than the solution."

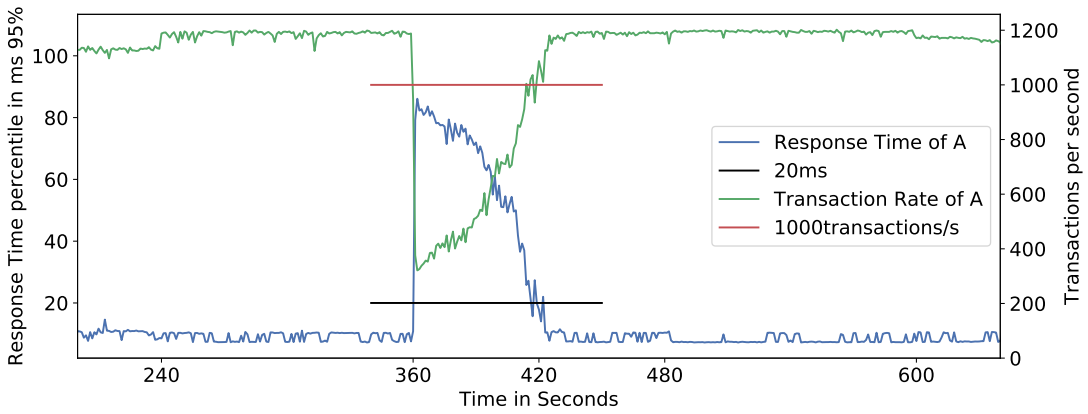


Figure 4.2: The quality of service of *A* is degraded when *C* boots.

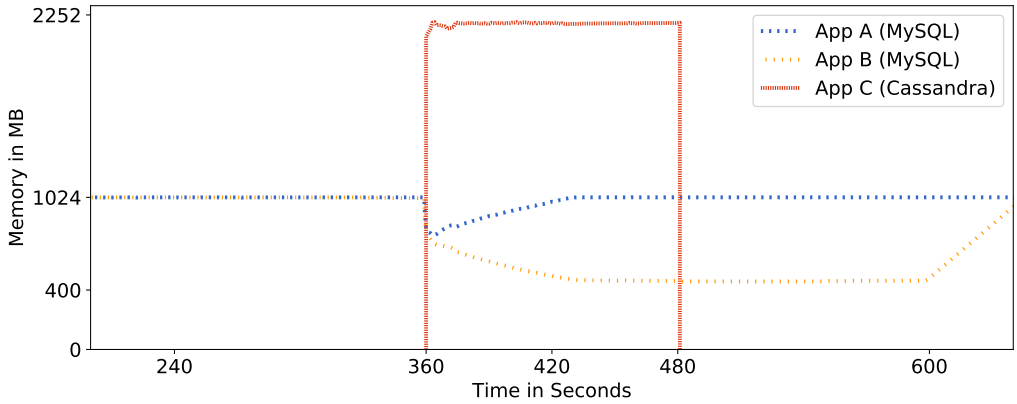


Figure 4.3: As *C* boots, memory is taken from the wrong container—i.e., *A*, the active container—when it should have been taken from *B* only.

memory so that only two of them can be active at the same time. *A* models an application during an event. It is active throughout the experiment $[0s, 840s]$ but takes $100s$ to warm up. *B* and *C* model applications that change their activity status, their resources are temporally multiplexed. *B* is active at the beginning $[0s, 240s]$ and becomes inactive in the middle $[240s, 600s]$, then it becomes active again until the end $[600s, 840s]$; *C* activates itself at the middle of the experiment $[360s, 480s]$, in the time frame in which *B* is inactive. When *A* and *B* are active, the benchmark is configured to generate as many requests as possible. When *B* deactivates and *C* activates, the benchmark is configured to send requests to neither *B* nor *C*, but it still generates requests to *A*. Given such conditions, we believe that *A* is obviously the most active candidate whose performance must be isolated during consolidation.

All the experiments in this thesis were done on an Intel(R) Core(TM) i7-4770 CPU @ 3.40 GHz with 2 memory banks HMT351U6EFR8C-PB of 4 GB and a Samsung SSD 840 of 128 GB. The version of Linux kernel was 4.6.0 compiled with gcc 4.8.4. *A* and *B* were given 2 cores, 1 GB

of memory and 12 MB/sec of bandwidth to the SSD. *C* was allowed to execute on *B*'s cores and had to reuse about 512 MB of *B*'s memory. The queries to MySQL were generated using Sysbench [144] and we set the Service-level Agreement (SLA) such that: *i)* 95% of the requests had to be executed in less than 20 ms, and *ii)* the transaction rate had to be at least 1000 transactions per second. This QoS is fairly achievable on our machine and better results have been obtained by Felter *et al.* when they compared VMs to containers on a 16 cores machine with 256 GB of RAM [17].

When the Cassandra application (App *C*) starts at time 360*s*, the QoS of *A* is degraded by a factor of 4 during one minute. The 95%tile response time of the queries of *A* does not respect the expected level of 20 ms and the transaction rate drops below 1000/sec (see Figure 4.2). Indeed, the memory of *B* is reclaimed and transferred to *C*, but despite being active, *A* loses about 256 MB of memory (see Figure 4.3). *A* is then forced to reload its data which causes more memory to be reclaimed; some from *B* and some from *C*. All these extra transfers explain the QoS loss in *A* during consolidation. The next section investigates why memory is erratically transferred during consolidation.

4.2.2 Consolidation without containers

We wanted to validate the hypothesis that the problem, i.e., the erratic transfers of memory during consolidation, was solely due to the fact that the applications were deployed in containers. Consequently, we crafted a micro-benchmark to model *A*, *B*, and *C* such that they could be easy to study out of containers. We replaced *A* and *B* with Filebench [81] processes that did not need isolation because we throttled them using Filebench's workload model language [64]. *C* was replaced with a simple program written in C which allocates memory through a single `malloc` call, accesses it in a loop and finally calls `free` at the end.

We repeated the scenario described in the previous section twice: first by deploying the applications in containers and then by deploying them without containers. The results are reported in Figure 4.4. With containers, we observed a QoS loss in *A* because its memory was collected along with *B*'s when *C* booted. The disturbance lasted for 40 seconds, and the reading throughput dropped from 700 MB/sec to 160 MB/sec. When deployed out of containers, the QoS of *A* was not impacted by the start-up of *C* because the only memory reclaimed was that of *B*[3].

4.2.3 Measuring consolidation errors

We believe that the problem of memory consolidation is not specific to Magency and that others may struggle with the same problem without noticing it. Actually, the first symptom of this problem is **not** a tangible degradation of performance, but instead, it is an overconsumption of disk bandwidth. Indeed, as long as *A* has enough disk bandwidth to cover up the errors of consolidation, it can quickly reload its data to compensate the mistakes

[3]The memory of *A* and *B* was measured with the `cache` counter and the memory of *C* with the `anon`.

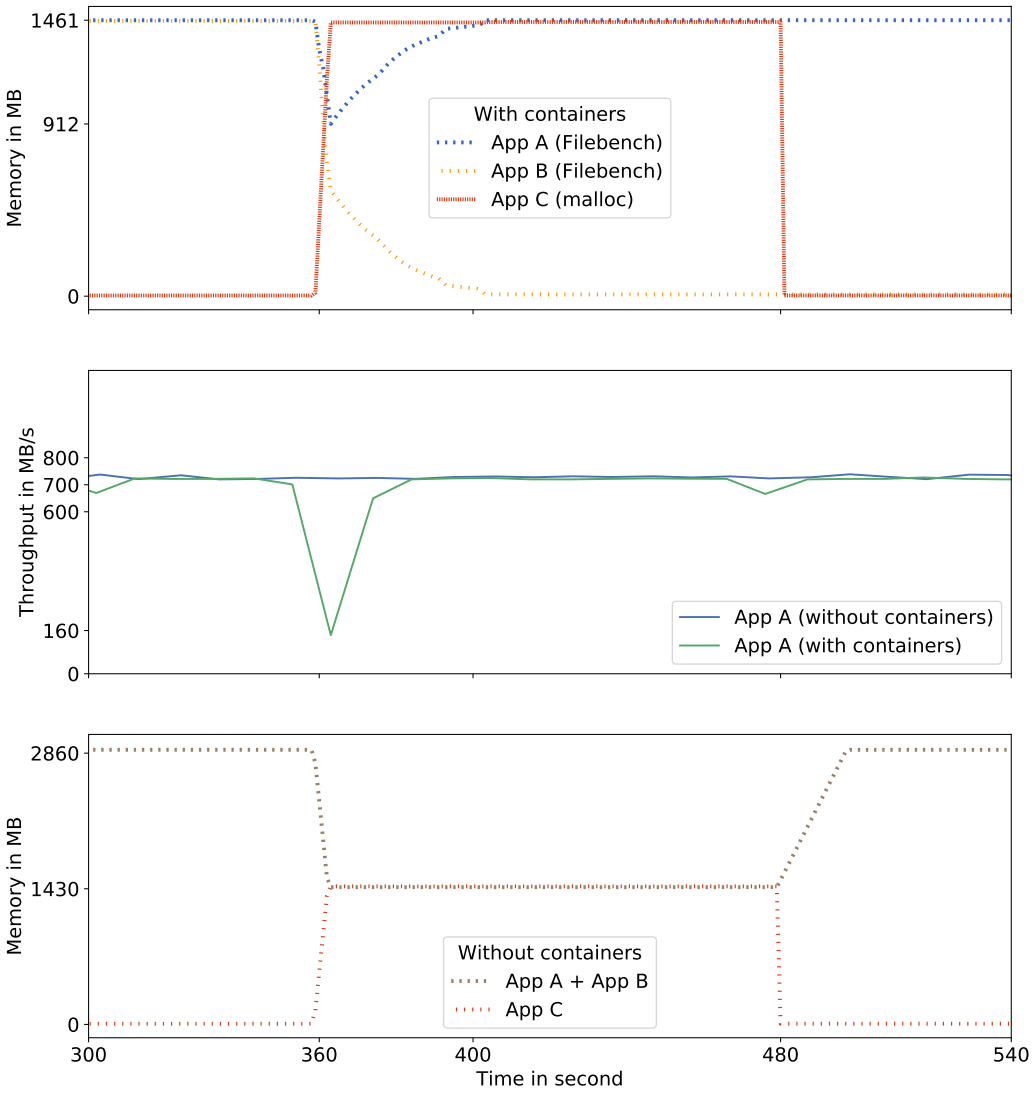


Figure 4.4: Running Filebench processes in and out of containers.

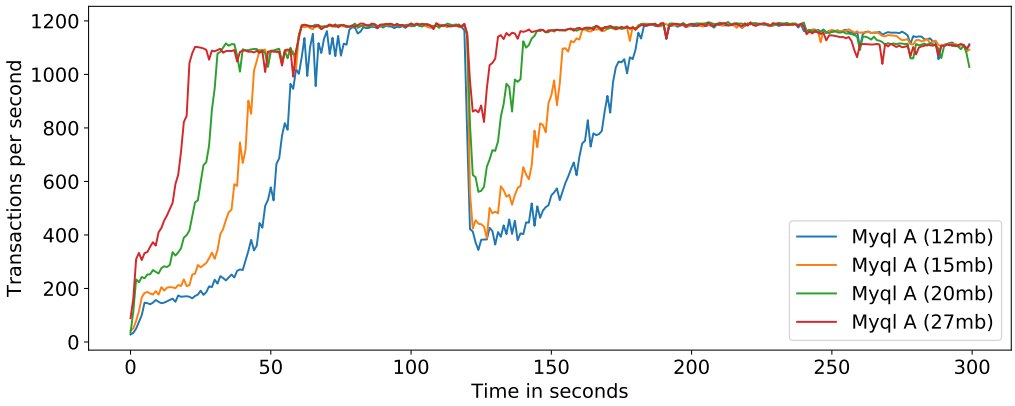


Figure 4.5: As the disk bandwidth increases, the performance drop disappears.

and prevent consequent performance degradations. Nevertheless, disk bandwidth over-consumption is a problem.

To show that higher disk bandwidths can prevent someone from discerning the problem of memory consolidation, we repeated the experiment in Section 4.2.1[4] and varied the disk bandwidth capacity of A from 12 MB/sec to 27 MB/sec. As shown in Figure 4.5, when the disk bandwidth increases, the performance drop disappears.

However, extra disk bandwidth should not be wasted during consolidation because it should be dedicated to waking up containers that wish to reload their data or to active containers with tremendous datasets that need to be streamed. Moreover, disks can sometimes be slow because they can be on a remote storage or because they can be spinning disks. Even if SSDs were used, the consolidation errors would severely impact their writing endurance.

We came to the conclusion that performance metrics or resource consumption metrics were not good enough to measure the consolidation errors. Therefore, we introduced two new page event counters per cgroup: `pglost` and `pgstolen`. These counters can capture consolidation errors independently of the disk bandwidth and therefore of the application performance.

- `pglost` is an estimation of pages lost, it counts the number of pages reclaimed in a cgroup because another cgroup has reached their common parent limit.
- `pgstolen` is an estimation of pages stolen, it counts the number of pages reclaimed by this cgroup in other cgroups because this cgroup has reached their common parent limit.

Thereby, if the `pgstolen` counter of a cgroup quickly follows its `pglost` counter, we can then consider that this cgroup incorrectly lost its memory because it quickly wanted to

[4]For practical reasons, we shorten the total time of the scenario.

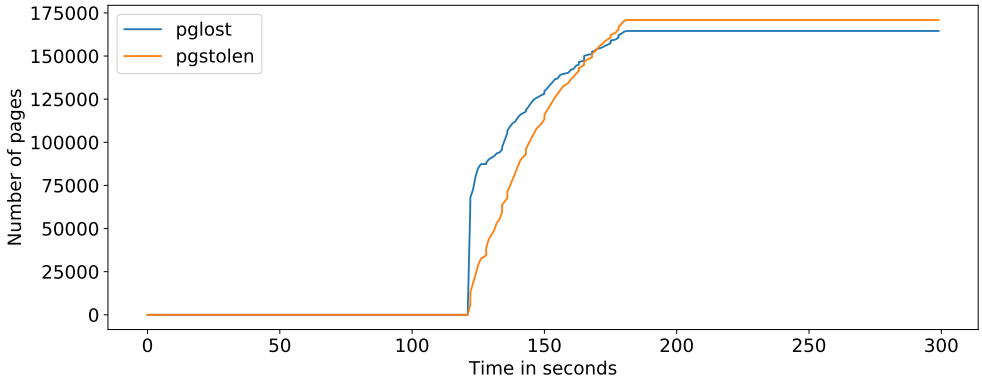


Figure 4.6: `pgstolen` and `pglost` in *A*.

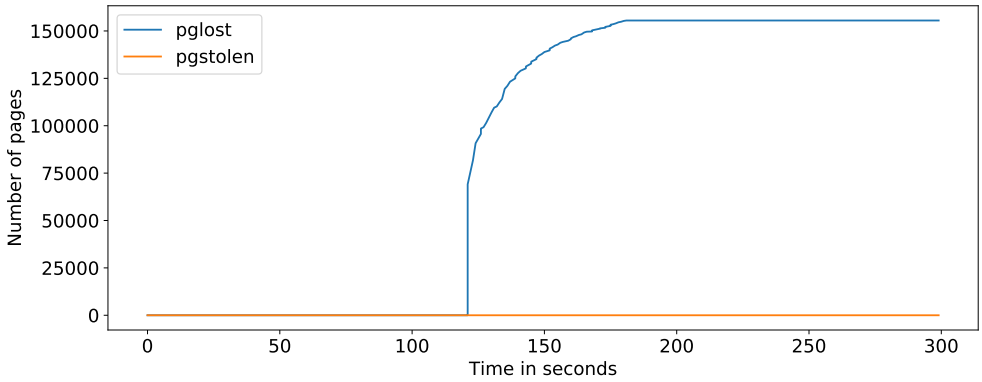


Figure 4.7: `pgstolen` and `pglost` in *B*.

recover it back by stealing pages to other cgroups. This behavior can, for instance, be observed in the previous experiment: despite the fact that *A* was given 27 MB/sec of disk bandwidth and that very little disturbance was observed in its transaction rate (recall Figure 4.5), we can observe in Figure 4.6 that its counters were very close to each other. On the other hand, as *B* was inactive, it did not try to steal back the pages taken from it (see Figure 4.7).

4.3 Lesson learned

We have defined a modeling framework in which we could study consolidation events. But we have discovered that during consolidation, Linux made errors that lead to QoS disturbance. These errors did not occur when we carefully removed the need for isolation. Thanks to what we learned in Chapter 3, we can provide an explanation:

When applications are deployed without isolation, they share the same cgroup, i.e., the same memory pools. These memory pools are complex (recall Figure 3.5) but they can be

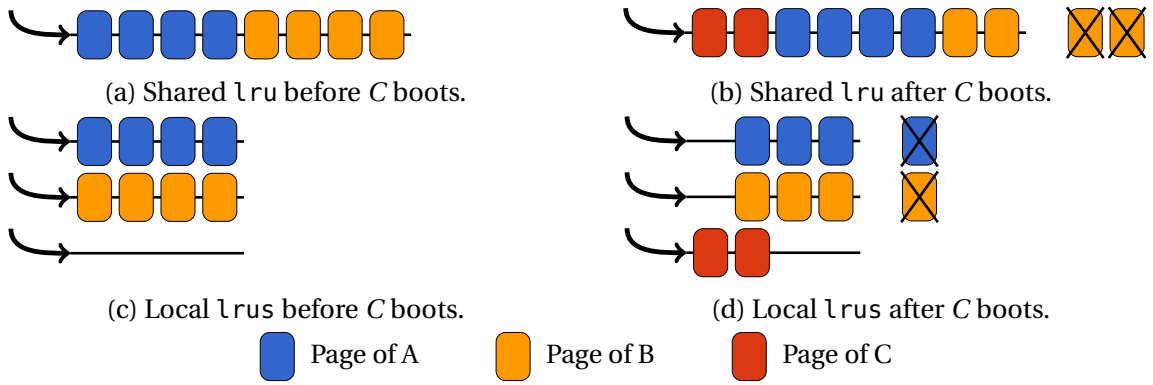


Figure 4.8: Evictions in `lrus`.

interpreted as simple page lists like the one in Figure 4.8a. This particular page list represents the state of the `lru` shared between *A* and *B* just before the boot of *C*. The pages of *A* are colored in blue, those of *B* are yellow and the red ones belong to *C*. Since *A* is more active than *B*, most of its pages will be more recently used than the pages of *B*. This is the reason why we represented the pages of *A* on the left and the pages of *B* on the right. Consequently, when *C* asks for memory and inserts its pages in the list, the PFRA correctly evicts the pages of *B* and keeps the pages of *A* (see Figure 4.8b).

Unfortunately, when isolation is required between applications, their pages are segregated in local lists (see Figure 4.8c). In a nutshell, the kernel traded the global recency order to provide good isolation between cgroups. But when pages are stored in different lists, their recency order can no longer be compared. As a result, the PFRA does not know that the pages of *B* are less recently used than those of *A* and when *C* boots, it chooses to evict pages from both *B* and *A* (see Figure 4.8d).

Based on this observation, we concluded that a preference order between the local lists of *A*, *B*, and *C* was necessary. In other words, the kernel has to know that the pages of *C* are more important than the pages of *A* which are more important than the pages of *B*. Thus, Chapter 5 will study if it is possible to distinguish an active container such as *A* from an inactive container such as *B*. Then, Chapter 6 will describe our kernel mechanisms that protects *A* from *B* when *C* asks for memory. Finally, the evaluation of Chapter 7 will use experimental setups slightly harder to consolidate than the one presented in this Chapter. Indeed, the inactive server will still receive requests, but they will either income at a slower rate or be configured to access only a small subset of the data.

CHAPTER 5

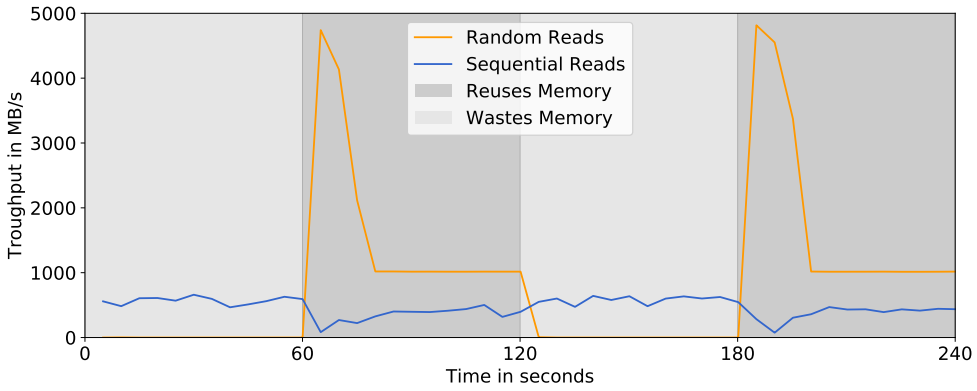
CAPTURING ACTIVITY SHIFTS

Before diving into the heat of the battle, it would be wise to study if it is possible to capture the activity shifts, and at what cost. Thus, in this chapter, we evaluate, independently of memory consolidation, if the *rotate ratio* and the *idle ratio* are metrics accurate enough to capture the shifts in activity.

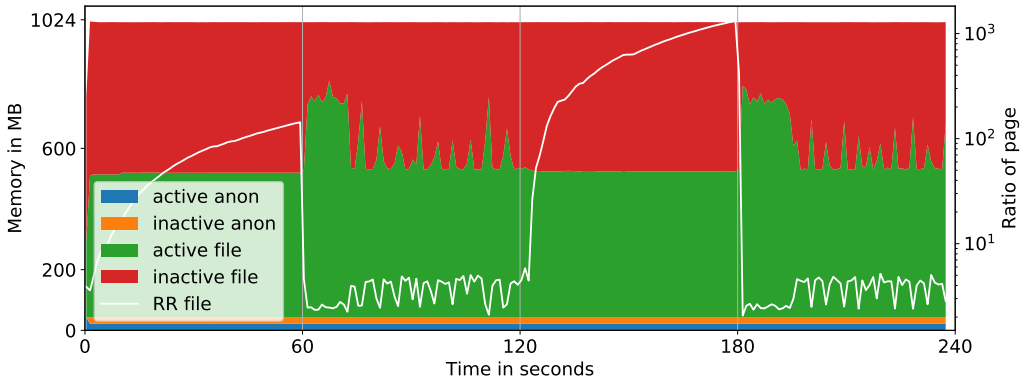
5.1 *Rotate ratio*: a `lru` dependent metric

In the case of out of memory workloads, the `lrus` are extensively used to track the page workingset. But if no such set is detected, the workload is assumed to be wasting memory because it is asking for pages that are used once and never used again (recall Figure 3.3). The *rotate ratio* (RR) measures this efficiency: in essence, it goes to one when the workingset of an application fits in memory and grows bigger when an application starts to waste memory. As explained in Chapter 3, when pages are accessed, they carry out a rotation to the head of the `lru` (from the `inactive_list`, into the `active_list`). The number of rotations is then compared to the number of pages scanned, i.e., the number of pages processed by the PFRA. Therefore, even if the RR is "freely" computed by the kernel, it can produce false negatives, especially when the PFRA is not triggered anymore.

In this section, we first show that RR detects obvious I/O patterns that waste memory; then we show how RR is used to balance the size of the `anon` and `file` `lrus`; and finally we show that RR can produce false negatives. To counteract these errors, we developed `force_scan`, a mechanism that refreshes the state of the `lrus`. This mechanism is later described in Section 6.1.



(a) Two phase types: One which wastes memory and one which reuses it.



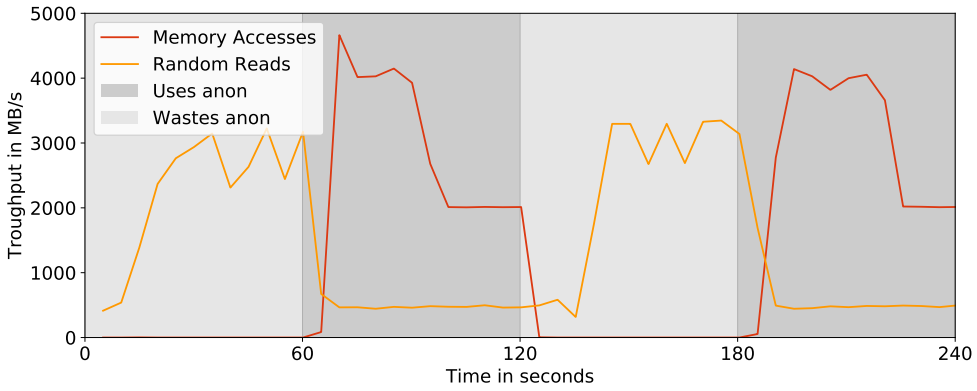
(b) The amount of active/inactive memory does not detect memory waste, but RR does.

Figure 5.1: *Rotate ratio* detects I/O patterns that waste memory.

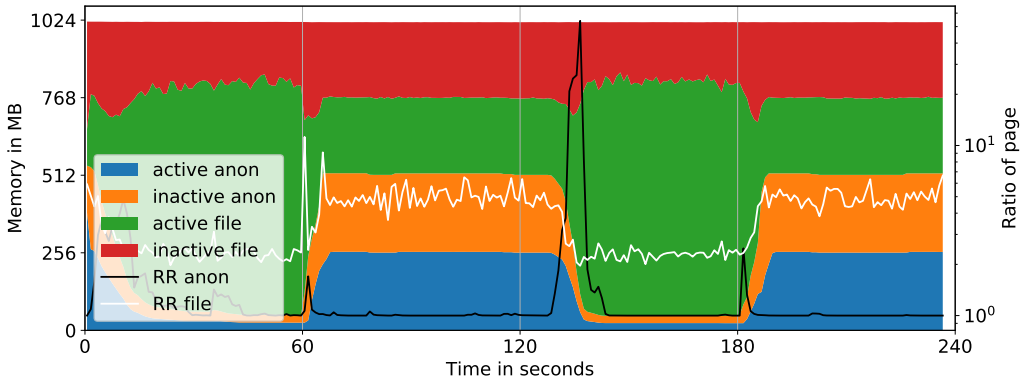
5.1.1 Detecting I/O patterns that waste memory with RR

To show that RR detects obvious I/O patterns that waste memory, we crafted a workload with Filebench [81, 64] that changes its behavior over time. The experiment span over four minutes and at each minute, the workload alternates its behavior: during the first and third minutes, memory is wasted; and during the second and fourth minutes, memory is re-accessed. To achieve this dual behavior, two types of threads are used. The first type loops without limitation and sequentially reads a file of 2 GB. As the cgroup is limited to 1 GB, the first type causes the workload to waste memory. On the other hand, the second type of threads is only activated during the second and fourth minutes. These threads randomly access the first GB of the file and therefore cause the workload to re-access its memory. Both access pattern throughputs are reported in Figure 5.1a.

Figure 5.1b reports the amounts of `active` and `inactive` memory of the `anon` and `file` `lrus`. The amount `anon` memory is very low because it is only used to buffer I/O by chunks of 1 MB. When the random threads wake up at time 60 and 180 seconds, they activate a lot of pages. Then, the small following spikes of activation are made by the single thread that



(a) Two phase types: One which does not use its anon memory and one which does.



(b) RR of the anon lru increases when anon memory becomes useless.

Figure 5.2: The RR allows the PFRA to balance the size of the anon and file lrus.

sequentially reads the file. However, during the phase where memory is wasted, the PFRA does not deactivate all the pages. This strategy allows the workingset of the random threads to persist in memory even if it is not accessed anymore. Therefore, in this case, the ratio of active/inactive memory cannot detect the waste of memory. However, we can observe that the RR of the file lru is close to one (below 10) when the memory is re-accessed and quickly grows to the hundreds in a few seconds when memory starts to be wasted at time 120. Moreover, at times 60 and 180, the RR immediately falls back to one as soon as memory becomes useful again.

In this experiment, the sequential thread keeps the lru alive when the random threads go to sleep. As a result, the RR is also kept up to date. But in Subsection 5.1.3, we show what happens if there is no more activity in the lru.

5.1.2 Balancing anon and file memory with RR

As explained in Chapter 3, the *rotate ratio* was introduced to balance the amounts of anon and file memory. To illustrate this mechanism, we crafted an experiment slightly different from the one presented the previous subsection. The workload is still composed of two types of threads, but the first type randomly accesses 1 GB of pages in a file without limitation while the second typed threads are only activated during the second and fourth minutes. The second type accesses 512 MB of mapped anonymous memory. Both access pattern throughputs are reported in Figure 5.2a.

Figure 5.2b reports the amounts of active and inactive memory of the anon and file lrus. At time 120s, the anon memory stops being accessed. A few seconds later, the *rotate ratio* of the anon lru increases. At this moment, the PFRA knows that the file memory is more useful than the anon memory and consequently begins to quickly swap out anon pages in favor of file pages. Then, as the size of the anon lru reaches the number of pages really in use, its *rotate ratio* falls back to one. Latter at time 180s, as they wake up, the threads are slowed down a little because they have to reload the page from the swap but they quickly recover the memory lend to the file lru.

We would like to achieve similar balancing property between cgroups, but as they do not scan their pages at the same rate, their *rotate ratio* might not be comparable, especially in the case described in the next subsection.

5.1.3 RR can produce false negatives

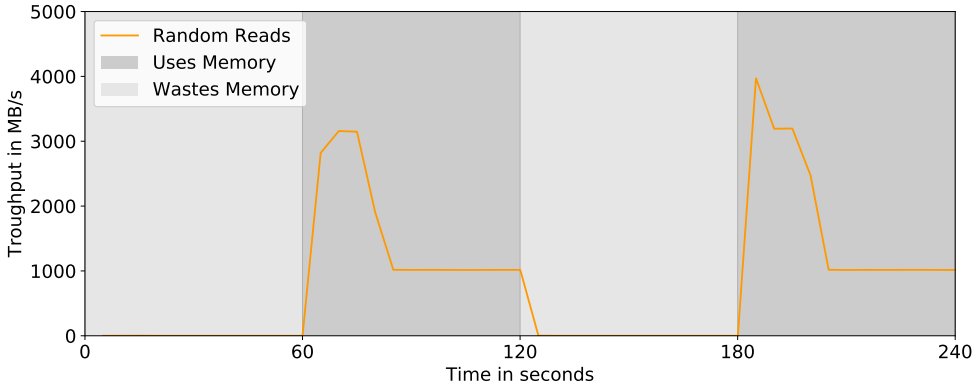
In the first subsection, we have shown that RR can detect obvious I/O patterns that waste memory. However, in this subsection, we show that once a workload stops paging in, the RR metric is not updated and becomes unreliable. To that end, the sequential thread is removed from the workload presented in Figure 5.1a to produce a new workload summarized in Figure 5.3a.

As shown in Figure 5.3b, without the sequential thread, the lru are not updated and the value of its RR does not increase during the third minute. As a result, we cannot deduce that the workload is wasting memory. Removing the sequential thread is an obvious means of showing the problem of using RR to balance memory between cgroups. More generally, the same problem can appear when cgroups scan their lru at different rates.

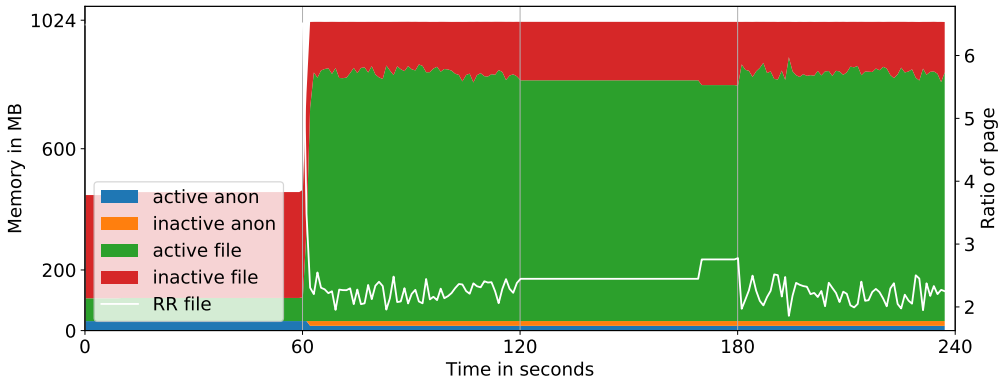
To solve this problem, the rate at which the lru is updated can be artificially increased with force_scan, a mechanism that we developed. As explained later in Section 6.1, this new mechanism allows programs from userspace to scan extra pages in a cgroup without freeing them. For instance, on this particular workload, we are able to detect that the memory is unused by simply scanning an extra MB of pages per seconds (see Figure 5.3c).

5.1.4 Additional force_scans cost CPU time and impact isolation

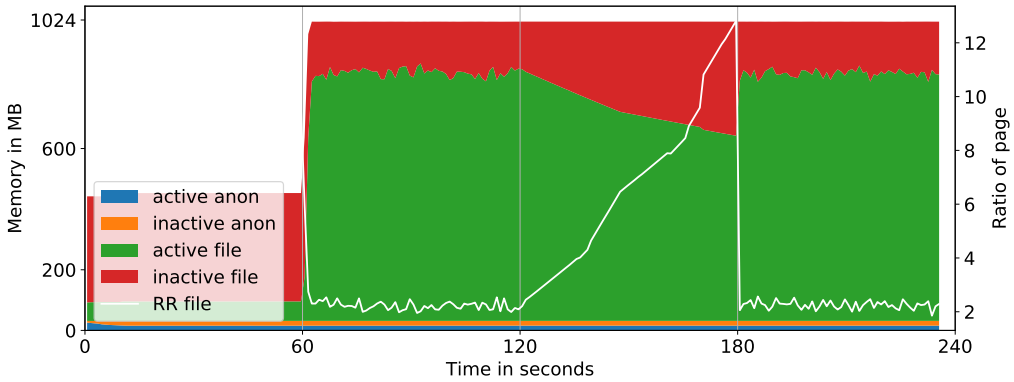
Ideally, if cgroups were to scan their pages at the same rate, just as the anon and file lrus of the same cgroup do, their RR would be comparable. Unfortunately, additional



(a) Two phase types with random threads only.

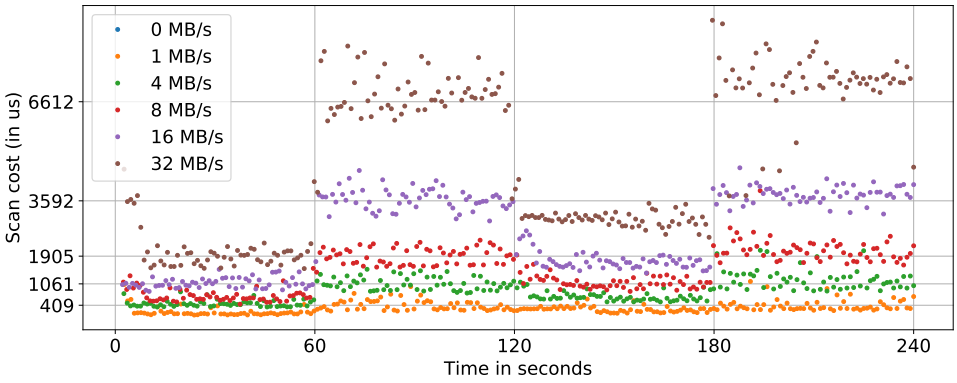


(b) The `lru` and its RR are not updated during the "waste memory" phase.

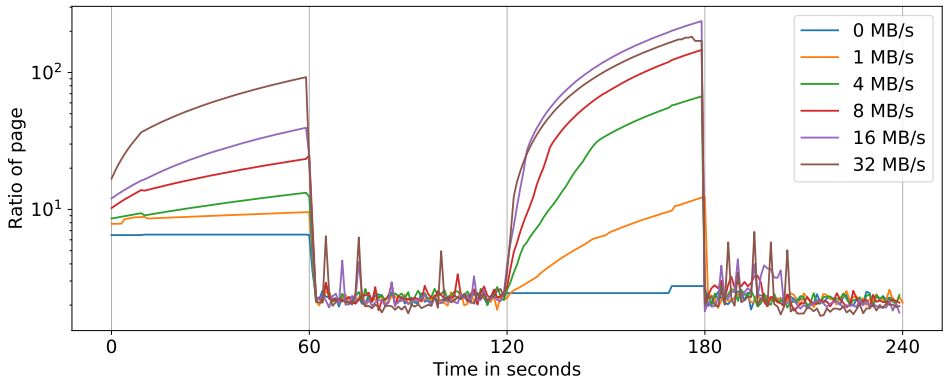


(c) The `lru` is artificially updated at an extra 1 MB/s rate.

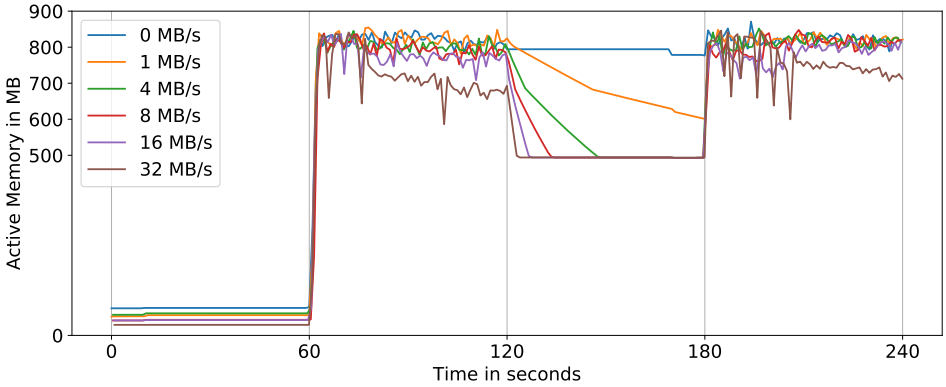
Figure 5.3: Extra page scans are sometimes necessary to confirm the RR values.



(a) Scan CPU costs measured with *ftrace*.



(b) RR grows faster with higher scan rates.



(c) Page Activation is impacted with higher scan rates.

Figure 5.4: Evaluation of additional memory scans.

force_scans cost CPU time and impact isolation.

To measure the CPU time of the force_scans, we used *ftrace* and repeated the experiment with different scan rates (1, 4, 8, 16 and 32 MB/s). The time spent in microseconds during every scan is reported in Figure 5.4a. We can observe that the cost is high during the second and fourth minutes because the scan has to acquire locks on the lru while it is already being updated by the workload.

With higher scan rates, the RR grows faster when memory is wasted, which leads to earlier detections (see Figure 5.4b). But scan rates higher than 16 MB/s start to influence the workload's ability to activate its pages (see Figure 5.4c).

5.1.5 Conclusion

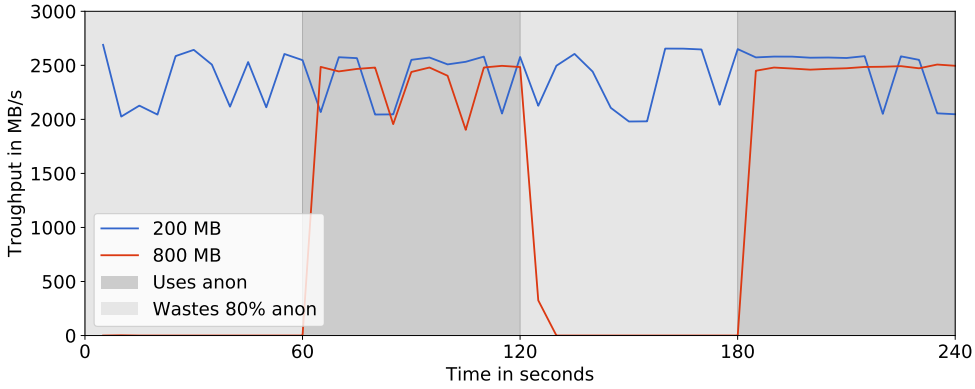
To conclude, we recommend the use of this metric to balance memory between cgroups since it is already present in the kernel for a similar purpose. Even if the cgroups do not scan their lru at the same rate, these rates could be readjusted with *force_scan*s. To restrain the impact of *force_scan*s on the quality memory isolation, they could be triggered in a best effort fashion only when an internal cgroup reaches its limit. This technic is used in one of our solutions (see Section 6.3.2).

5.2 *Idle ratio*: a lru independent metric

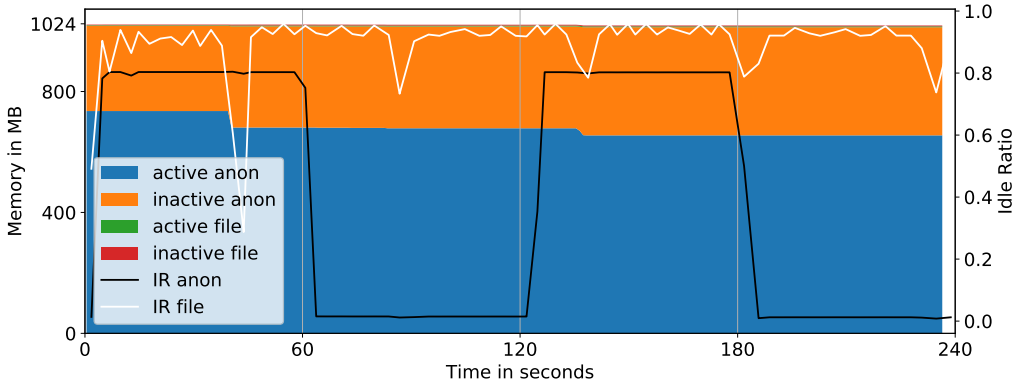
In the case of in-memory workloads, the lrus are almost useless because there is no need to track the page workingset since everything fits in memory. However, once in a while, these workloads can stop accessing part of their memory. Thereby, if a page has not been accessed for a given amount of time, it should be tagged as idle and reused for another purpose in another cgroup for instance.

The *idle ratio* (IR) detects this behavior: in essence, given a scan period, it goes to one if all pages have been accessed in the scan period or goes to zero if no pages have been accessed during the period. IR requires a specific kernel compile configuration [107, 99] that adds two extra bits per page (the idle and the young bits) and an API to interact with them. In contrast to RR, IR is not "freely" computed by the kernel and has to be computed by a dedicated monitoring application in userspace with a two-step procedure. First, the monitor has to mark the desired pages to track as idle, but since the API is lru independent, there is little incentive to monitor a subset of cgroups in particular. Then, every time the kernel sees an access to the page, it clears the idle bit (recall mpa in Figure 3.3). Finally, after a period of time, the monitor can come back to see if the page is still marked as idle. The idle page tracking mechanism also interacts with the accessed bit of the PTE. Therefore, to avoid interferences with the PFRA, a second bit—the young bit—is used to remember the value of the accessed bit if the monitor wants to clear it.

In this section, we provide an in-memory workload example that shows that IR can accurately monitor the set of idle pages; and finally we show the trade-offs between CPU time



(a) Two phase types: One with 80% of idle memory and one with 0%.



(b) The amount of active/inactive memory is static, but IR detects memory waste.

Figure 5.5: *Idle ratio* tracks down unused pages of in-memory workloads.

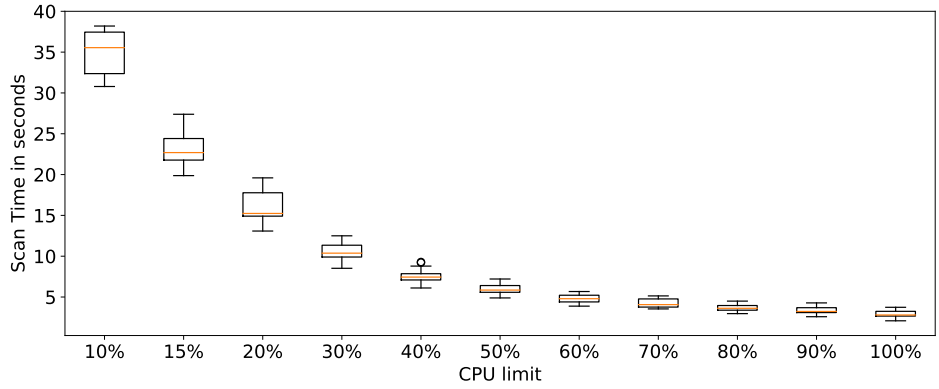
cost and accuracy. IR can also be applied in the case of out of memory workloads, but in-memory workloads are harder to monitor because they usually involve direct memory accesses to mapped pages which are not caught by the kernel for performance reasons.

5.2.1 IR accurately monitors the set of idle pages

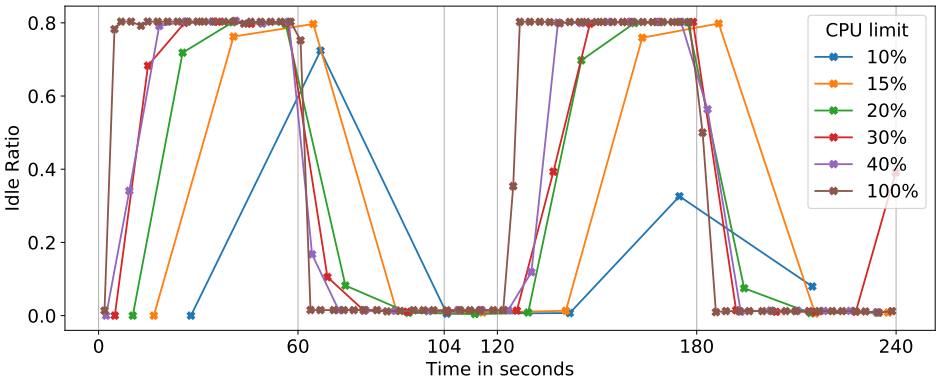
To show that IR can accurately monitor the set of idle pages, we crafted a workload similar to the ones presented previously with Filebench[1]. We used two types of threads: one which always accesses its 200 MB of memory and the other which accesses its 800 MB only during the second and fourth minutes. Both threads never access data on disk and are therefore in-memory workloads. Their throughputs are reported in Figure 5.5a.

Figure 5.5b reports the amounts of `active` and `inactive` memory of the `anon` and `file` `lrus` and their IR. There is very little memory in the `file` `lru` and as most of it is unused, the `file` IR is close to one. Most of the memory is used to store `anon` pages and since this

[1]We modified the *hog flowop* to access all the memory allocated by a thread.



(a) Scan Time of IR as a function of CPU usage.



(b) With higher scan time, IR lags behind the real workload activity.

Figure 5.6: Evaluation of the trade-offs between CPU time and IR's accuracy.

workload fits in-memory, there is almost no movement of pages between the `active` and `inactive` lists of the `anon lru`. However, the IR of the `anon lru` is able to measure that during the first and third minutes, 80% of the pages are idle.

5.2.2 Trade-offs between CPU time cost and IR's accuracy

V. Davydov built a smart tool in userspace—i.e., idlememstat—to take advantage of the idle page tracking kernel compile configuration [106]. By default, the tool checks then marks pages by chunks of 32k pages; and spreads the total scan over a period of 300 seconds. The reason behind this default behavior is to avoid CPU bursts. But in practice, the values will depend on the total amount of memory available on the machine, how that memory is used by the workloads and the speed at which the user wishes to detect idle memory.

We evaluated the speed at which memory can be scanned on the same experiment presented in the previous subsection. Idlememstat was tweaked to ignore its scan throttling option and to report the time it takes to complete a single scan. Then, we repeated the

	RR	IR
Can be freely computed	Yes	No
Does not interfere with the `lru`	No	Yes
Does not have a memory footprint	Yes	No
Can monitor specific containers	Yes	No
Can estimate workingset size	No	Yes

Table 5.1: Comparing the *Rotate ratio* to the *Idle ratio*.

experiment and throttle its CPU time consumption using the `cpu` cgroup. The scan times are reported in Figure 5.6a. Given a full CPU core (100%), the scans take on average 2.9 seconds, and given 10% of CPU time, they take on average 34.9 seconds.

Figure 5.6b reports the values output by idlememstat for some CPU time limits. When the scan time increases, there are fewer sample points collected but most importantly, the IR begins to lag behind the real workload activity. For instance, at time 66s, with 10% of CPU time, the monitor detects that 60% of memory is idle when actually, 0% is idle. Moreover, since it takes on average 34.9 seconds to sample another point, it only discovers it later, at time 104s.

5.2.3 Conclusion

Despite its CPU cost and its two extra bits per page, the idle page tracking is very useful especially in the case of in-memory workloads. As it does not interact with the `lru`, this mechanism preserves the quality of the memory isolation as opposed to the `force_scan` mechanism. But on the downside, it cannot be used to monitor a single cgroup in particular, the whole machine has to be monitored. Overall, we recommend this metric over the RR because it can also be used to size the `hard` memory limit of containers.

5.3 Conclusion

We have shown that the *Rotate ratio* and the *Idle ratio* can capture memory activity shifts. A summary of their properties is provided in Table 5.1. In the next chapter, we will use the *Rotate ratio* and the *Idle ratio* to build a `reclaim_order` between containers. However, these metrics are not the only one able to track activity; kernel events such as page demands and activations can also do the trick. Unfortunately, due to lack of time, we did not showcase the evolution of these event counters over time under workloads that change their activity.

CHAPTER 6

SUSTAINING ISOLATION OF CGROUPS

In the previous chapter, we acknowledged that capturing the shifts in memory activity levels is possible. In this chapter, we assume that cgroups can be ordered according to their activity level and we describe the kernel modifications required to build an optimal solution, based on this assumption, that would consolidate memory without errors. Unfortunately, the optimal solution is infeasible in practice because to design such a solution, one would have to be able to predict the future. The aim of this chapter is to propose solutions that can guess the activity levels. The solutions that we devised can be classified into two general approaches: the metric-driven one and the event-driven one. But before getting to the heart of the matter, we first introduce our `force_scan` mechanism.

6.1 Refreshing the `lrus` with `force_scan`

The `lru` data structure in the Linux kernel has a specific design which reduces unnecessary page movements in it. As a result, when applications can fit their data in memory, the state of the `lru` does not change and all the metrics natively computed by the kernel are not updated. This is the case for the ratio of active pages, the vmpressure but also for the *rotate ratio* (as shown in Section 5.1.3). Once they stop being updated, if the applications stop using part of their memory, the metrics will not detect it (recall Figure 5.3b).

To solve this problem, we implemented `force_scan`, a mechanism that allows monitoring programs in userspace to artificially increase the rate at which pages move in the `lrus`. We provide a new `force_scan` file per cgroup in the `sysfs` API: when a number of bytes is written to the file, the kernel will try to scan the corresponding number of pages in the `lrus` of the cgroup. The `force_scan` file is different from the existing `force_empty` file which tries to free all the pages of the cgroup.

```
1  Function PFRA(nr_to_reclaim, root)
2      repeat                                                    /* 2664 */
3          foreach zone do                                       /* 2614 */
4              foreach cgroup ∈ root do                          /* 2423 */
5                  w = get_scan_count()                          /* 2209 */
6                  foreach lru do                                /* 2241 */
7                      if inactive_list_is_low(lru) then         /* 1944 */
8                          pages = isolate_lru_pages(w,lru.active) /* 1795 */
9                          foreach p ∈ pages do
10                             page_referenced(p)                /* 1825 */
11                         end
12                         move_active_pages_to_lru(pages)       /* 1859 */
13                     end
14                     pages = isolate_lru_pages(w,lru.inactive) /* 1602 */
15                     tokeep = list(), tofree = list()
16                     foreach p ∈ pages do
17                         if page_referenced(p) then             /* 798 */
18                             tokeep.add(p)                     /* 1224 */
19                         else
20                             /* unmap(p) writeback(p) swap(p)...        */
21                             tofree.add(p)                     /* 1204 */
22                             nr_reclaimed++                    /* 1198 */
23                         end
24                     end
25                     mem_cgroup_uncharge_list(tofree)          /* 1228 */
26                     free_hot_cold_page_list(tofree)           /* 1230 */
27                     putback_inactive_pages(tokeep)            /* 1638 */
28                 end
29                 if nr_reclaimed >= nr_to_reclaim then
30                     mem_cgroup_iter_break()                   /* 2448 */
31                 end
32             end
33         end
34     until (nr_reclaimed < nr_to_reclaim) && (priority-- > 0)
35     return nr_reclaimed
```

Figure 6.1: Function do_try_to_free_pages of the PFRA.

By now, we've done our best to avoid presenting code in this document. But some pseudocode would be more than welcome to abstract the complexity that we had to face to develop code in the kernel. Figure 6.1 presents a simplified version of PFRA that the reader is already familiar with.

To implement `force_scan`, we decided to run a modified version of the PFRA that never triggers the functions (at Lines 25 and 26) that uncharge and free pages but instead, always triggers the function (at Line 27) that keeps the pages. To reach that end, we modified the outcome of the function (at Line 17) that tests if a page was recently accessed. If this function decides that the page can be reclaimed, we override its return value with the value which asks to keep the page.

`force_scan` also modifies the function (at Line 5) that computes the number of pages to scan in the `anon` and `file lrus` so that they are both scanned (recall Section 3.4.1). In our case, we did not need the ratio of active pages to fall to 0, but if it were necessary, `force_scan` could also override the return of the function (at Line 7) to remove the protection of the `active_list`.

`force_scan` keeps the `lru` metrics up to date but it has many disadvantages: it takes locks (at Lines 8 and 14), displaces the page position (at Lines 12 and 27), modifies the access bit in the PTEs (at Lines 10 and 17) and has a CPU cost (recall Section 5.1.4).

6.1.1 Conclusion

The *rotate ratio* needed a mechanism to keep synchronized the *lru*s of different cgroup. The development of `force_scan` fulfilled this requirement by faithfully emulating the behavior of the PFRA but without actually freeing the memory. Unfortunately, this mechanism has a heavy cost and interferes with the PFRA. To mitigate these disadvantages, we incorporated *force_scan* in another solution, on demand, i.e. only when an internal cgroup reaches its limit (see Section 6.3.2).

6.2 Building `opt`: a relaxed optimal solution

In this section, we will define `opt` as the goal that we will try to reach in the next sections with our metric and event-based solutions. But first, let's define **`inf`** as the case where we have an infinite amount of memory at our disposal. At any time, `inf` delivers the best performance possible because there is no need to transfer memory to avoid waste. If `opt` were to provide similar performance to `inf`, it would have to transfer pages before they were requested to hide the delay of the transfers. Indeed, transferring a page takes time because data has to be swapped in and swapped out of memory to disk. But this definition of `opt` is almost impossible to achieve because it would have to remember and prefetch all evicted pages.

A realistic alternative can be achieved if we relax our expectations in terms of performance isolation. We define **`opt`** as the solution that *i)* only steals pages from the most inactive

```
1  Function ReclaimPolicy(nr_to_reclaim, root)
2  |  /* decisions[i] = (cgroup[i],to_reclaim[i])                    */
3  |  decisions = kmalloc(root.nr_children)
4  |  init_and_sort(decisions, mem_excess, cmp_excess, root)
5  |  foreach d ∈ decisions do
6  |  |  if nr_reclaimed < nr_to_reclaim then
7  |  |  |  nr = nr_to_reclaim - nr_reclaimed
8  |  |  |  nr = min(nr, d->to_reclaim)
9  |  |  |  nr_reclaimed += PFRA(nr, d->cgroup)    /* Reclaim one cg */
10 |  |  else
11 |  |  |  break
12 |  |  end
13 |  end
14 |  kfree(decisions)
15 |  if nr_reclaimed < nr_to_reclaim then
16 |  |  nr = nr_to_reclaim - nr_reclaimed
17 |  |  nr_reclaimed += PFRA(nr, root)        /* Do classic reclaim */
18 |  end
19 |  return nr_reclaimed
```

Figure 6.2: A reclaim policy that always obey to the `soft_limits`.

containers, and *ii)* establishes transfers at the very last moment, i.e., only when the active containers request pages. The performance delivered with this definition of `opt` is not as good as `inf` because there will always be a warm-up phase when a server wakes up, but the assumption of consolidation is that the time spent reloading data is worth the cost of having extra memory. In Chapter 7, we show that `inf` is able to do 7% to 36% more transactions than `opt` because it does not have to reload data during the warm-up phases, but in order to do so, `inf` has to consume 50% more memory than `opt`.

6.2.1 Applying `soft_limits` at all levels of the hierarchy

As explained in Section 4.1.1, we implemented `opt` by dynamically readjusting the `soft_limits` when the activity shifts: when a container deactivates, a dedicated system daemon (as suggested in Section 3.4.2) sets its `soft_limit` to zero; and when a container reactivates its *soft_limit* is set back to the maximum. However, the current implementation of Linux does not apply `soft_limits` at all levels of the hierarchy; they are enforced only at the top of the hierarchy, i.e., when the whole machine is running out of memory. The cgroups are period-

ically ordered in a single red-black tree according to their memory excess. As introduced in Section 3.2.2, the memory excess is the difference between the current consumption and the `soft_limit`.

To apply `soft_limits` at all levels of the hierarchy, we did not use a red-black tree for every internal node; we chose a simpler approach. In the logic of the memory cgroup controller[1], when an internal cgroup[2] reaches its limit, we reroute the call, to the PFRA, to our new function `ReclaimPolicy` depicted in Figure 6.2. In `ReclaimPolicy`, we allocate an array of decisions (at Line 3). Each decision contains a pointer to a cgroup child of `root` and also contains the maximum amount of page to reclaim in the cgroup (in this case, this amount is computed by `mem_excess`). At Line 4, the array is initialized and sorted according to a compare function (in this case, the first cgroups are the ones the most exceeding their `soft_limit`). Then, the PFRA is carefully triggered on one cgroup at a time until enough pages have been reclaimed. Finally, the array is freed and the classic call to the PFRA is triggered if we did not reclaim enough pages. This final call to the PFRA reclaims pages in any child cgroup because the "for each" loop at Line 4 of Figure 6.1 has no preference order and breaks at Line 30 when enough pages have been reclaimed.

However, *soft_limits* are not ideal to implement `opt` because if there are multiple inactive containers, it would be better to steal pages from the container which will ask them back at the latest. Even if *soft_limits* can order containers by memory excess, it would be far-fetched to use this mechanism to emulate a temporal order between containers.

6.2.2 Order cgroups by activity levels with `reclaim_order`

In Section 4.3, we learned that the key to successfully preserve QoS during consolidation phases is to correctly identify and reclaim unaccessed pages first, before considering useful pages. Thus, inside a single list, LRU attempts to approximate Belady's optimal algorithm which evicts the page whose next access will occur the farthest in the future [7]. Unfortunately, the temporal order between pages is lost when they are stored in different `lrus` (recall Figure 4.8). However, previous work has shown that the `lrus` have to be strictly disjointed to preserve an isolated behavior during the steady phases (recall Section 3.3.2 and Figure 3.1).

Fortunately, all is not lost, because in our model we know that the unaccessed pages are in the inactive containers and that the useful pages are in the active containers (recall Section 4.1.1); but there is no way to express the activity order with the current implementation of the kernel. If there were a way to express it, the PFRA could then reclaim the least recently used pages in the hierarchy because they are the least recently used pages of the least active cgroup.

Our proposal is `reclaim_order`, a reclaim policy that allows any kind of total order between containers. Given such a policy, `opt` can express a temporal order between containers: the container the less likely to ask back its pages in the nearest future must lend

[1] Described in `try_charge @ mm/memcontrol.c`.

[2] In Figure 6.2, the internal cgroup is referred to as `root`.

its memory to the container which would ask back its page the earliest. In this thesis, we have only considered the case of ordering containers according to this definition of "activity level", but we could also consider any kinds of fixed or dynamic priorities. For instance, fixed priorities could be used as a fail-safe to protect the most important containers when all containers are asking for memory. On the other hand, dynamic priorities could be used to enforce fairness. For instance, the swap token algorithm [24] gives swap immunity to a single process to ensure that it had the chance to load all its pages. This algorithm could be generalized to cgroups by periodically changing the priority order and thus giving immunity to one cgroup at a time.

6.2.3 Stacking generic policies

`Reclaim_orders` and `soft_limits` are two of many possible policies that could guide the kernel during memory consolidation. To allow all of these policies to express themselves at the same time, we implemented a stack of generic policies: when the current policy fails at reclaiming enough pages, the next policy, which has to be stricter than the current one, is applied to reclaim more pages. As shown at Lines 5 and 20 of Figure 6.3, this scheme is repeated until enough pages have been reclaimed. We also incorporated the `force_scan` mechanism in the design at Line 16: when enough pages have been reclaimed, the policy can choose to `force_scan` some amount of page in the remaining cgroups that did not lose pages (recall Section 5.1.4). A policy is characterized by three functions which are used to initialize the decisions at Line 6:

- `reclaim` computes the maximum number of pages to reclaim in a cgroup,
- `cmp` compares two cgroups to build the decision order, and
- `scan` computes the maximum number of pages to `force_scan` in a protected cgroup.

Thus, `reclaim_orders` can simply be expressed as follows:

- `reclaim` returns the current consumption of the cgroup unless the user did not specify its `reclaim_order`.
- `cmp` compares a new `reclaim_order` field attached to the cgroup structure. This field is accessible from userspace through the `sysfs` API.
- `scan` returns 0.

As `reclaim_order` is more restrictive in terms of pages to reclaim than `soft_limit`, the former can only be stacked after the latter.

At this point, our prototype only supports the modification of some parameters of the policies at runtime (e.g., the `reclaim_order` field). The policies described in the next sections are statically stacked at compilation. For now, the prototype does not support the creation,

```
1  Function ReclaimPolicy(nr_to_reclaim, root)
2      /* policies[i] = (reclaim[i], cmp[i], scan[i]) are functions   */
3      /* decisions[i] = (cg[i],to_reclaim[i],to_scan[i])             */
4      decisions = kmalloc(root.nr_children)
5      foreach policy ∈ root.policies do
6          decisions = init_and_sort(decisions, policy, root)
7          foreach d ∈ decisions do
8              if nr_reclaimed < nr_to_reclaim then
9                  nr = nr_to_reclaim - nr_reclaimed
10                 nr = min(nr, d->to_reclaim)
11                 nr_reclaimed += PFRA(nr, d->cgroup)
12             else if d->scan then
13                 /* Policy wants to scan the protected cgroups        */
14                 nr = nr_to_reclaim
15                 nr = min(nr, d->to_scan)
16                 FORCE_SCAN(nr, d->cgroup)
17             end
18         end
19         if nr_reclaimed >= nr_to_reclaim then
20             break
21         end
22     end
23     if nr_reclaimed < nr_to_reclaim then
24         nr = nr_to_reclaim - nr_reclaimed
25         nr_reclaimed += PFRA(nr, root)       /* Do classic reclaim */
26     end
27     return nr_reclaimed
```

Figure 6.3: Stacking generic policies.

insertion, and removal of policies in the stack at runtime. But in future works, these features could be achieved at runtime with BPF and by attaching extra data to cgroup structures. Users could then describe custom policies such as incremental `soft_limits`, i.e., a list of decreasing limits for each cgroups. Moreover, each internal node of the cgroup hierarchy could have its own private stack of policies as suggested at Line 5 of Figure 6.3.

6.3 Guessing the activity levels

By now, we have provided new core kernel features: `force_scan`, `reclaim_order` and policy stacking. These features will allow us to implement multiple solutions. In Section 6.3.1, we describe solutions that guess activity levels from userspace and then communicate the information to the kernel through the `reclaim_order` API. Then, in Section 6.3.2, we describe solutions that guess the activity levels directly in kernelspace.

6.3.1 A Metric-driven approach to predict activities

The first approach attempts to predict the activities based on any previously observed metrics. Using this approach, a dedicated daemon in userspace collects metrics on a time window and communicates the resulting activity order to the kernel through the `reclaim_order` API. The assumption is that the activity order induced from the metrics will remain the same during the next collection interval.

Which metrics detect activity levels?

The metric-driven approach is very flexible because any kind of metric can be used to guess the activity levels. In Section 4.1.3, we first suggested using the time interval between two requests as an activity metric. However, application metrics such as throughput and response time are not always available to Cloud providers. Moreover, resource metrics such as CPU time and network bandwidth do not always correlate with memory needs. AI algorithms could be used to predict memory activity based on external metrics such as the ones aforementioned; but we did not have datasets from production to learn from.

We decided to only study the two major memory-related metrics presented in Chapter 5: the *rotate ratio*, and the *idle ratio*. Other memory-related metrics exist, but we did not study them: memdelay because it is not yet in the mainline, vmpressure because it is not reliable according to peers, TLB miss rate [18] because it does not account unmapped accesses, `/proc/[pid]/clear_refs` [77] because it is less powerful than the *idle ratio*, PAPI [36] since performance hardware counters have to be multiplexed.

The dilemma of delaying the metric or interfering with the workload

Despite its flexibility, the metric-driven approach has the complexity of having to size the time window. On the one hand, if the time window is too large, the daemon might miss

consolidation events. On the other hand, if the time window is too small, the metric computation might interfere with the workload. Moreover, independently of workload interferences and consolidation events, if the time window is too short, then all cgroups might appear inactive. Likewise, if the time window is too long then all cgroups might appear active. The ideal time window has to detect some as active and others as inactive.

Instead of sizing the time window, we chose to size CPU time consumption of the solution with CPU cgroup quotas and cpushares. As the *rotate ratio* and the *idle ratio* do not have the same update mechanisms, sizing the CPU consumption provides a fair comparison method for both metrics. In Chapter 7, we explored several ways of spending the extra CPU time required by metric based solutions. First, we tried to spend the least amount of CPU time (see `rr1%`). Then we assumed that all idle CPU time should be invested into updating the metric more frequently in the hope of improving the solution's reactivity (see `rrs` and `ir`). Finally, we decided to spend a fixed amount of 10% of CPU but due to lack of time, we did not test other values (see `rrs10%` and `ir10%`).

The following subsections describe our five metric-based solutions: `rr1%`, `rrs`, `rrs10%`, `ir`, and `ir10%`. To control the cost and interferences of these solutions, we studied them without memory consolidation and compared them to `inf` (see `irrs`, `irrs10%`, `iir`, and `iir10%`).

Solutions based on the Rotate Ratio

The acronym **`rr1%`** stands for "**r**otate **r**atio". It uses a daemon in userland, throttled to consume 1% CPU, to order the activity of cgroups according to the *rotate ratio* metric. This configuration does not trigger additional scans even if the *rotate ratio* value might be outdated.

The acronym **`rrs`** stands for "**r**otate **r**atio and **s**can". It uses a daemon in userland, throttled only by its cpushares, to scan the lru of cgroups and order their activity according to the *rotate ratio* metric.

`rrs10%` is similar to `rrs` but in addition to cpushares throttling, its daemon cannot use more than 10% of CPU.

Solutions based on the Idle Ratio

The acronym **`ir`** stands for "**i**dle **r**atio". It uses a daemon[3] in userland, throttled only by its cpushares, to mark pages idle, check if they are still idle and order the activity of cgroups according to the *idle ratio* metric.

`ir10%` is similar to `ir` but in addition to cpushares throttling, its daemon cannot use more than 10% of CPU.

[3] Idlememstat [106] was modified to accurately count pages that are not idle.

6.3.2 An Event-driven approach to react to activity changes

The second approach attempts to react as soon as possible to any instantaneous kernel event that suggests that the cgroup is active. In contrast to the monitoring of metric-based solutions which basically spin and wait for a change to occur in the activity order, this approach avoids the busy wait by hooking light operations to kernel functions. To reach that end, the approach uses a single global clock and operates on last event timestamps to track the most recently active cgroup. This method guarantees to produce a coherent total order on cgroups regardless of the rate at which the consolidation events occur. Therefore, even if all cgroups are active at a macroscopic level, there will always be a least recently active cgroup at the microscopic level. Moreover, even if the order produced is wrong, the solution will quickly learn from its mistake and promptly provide a correction thanks to its reactivity.

Which events should be considered?

There are many event types that could reveal the activity of a cgroup. Obviously, "page accessed" is the first type of event that comes to mind, but we did not use these events because they have two major disadvantages. First, they are too frequent and would cause the overhead of the hooks to become significant. Moreover, these events do not reflect the distribution of the accesses, i.e., if a cgroup does a lot of "page accessed" it does not mean that all its pages are accessed because the event does not account for distinct accesses.

The first type of events that we decided to track are "page demands" [12]. When a cgroup asks for more memory[4], the `hard_limit` page counters have to be hierarchically incremented. Therefore, we considered that adding another atomic operation to manipulate the clock and the timestamp should not hurt the performance. "Page demands" are not as frequent as "page accessed" but more importantly they reveal *i)* a change in the distribution of accesses because they are misses and *ii)* an intention to consume more memory.

The second type of events that we decided to track are "page activations". When accesses to a page in the `inactive_list` are detected[5,6], the page is "activated" and rotated in the `active_list`. These rotations are batched because they are expensive. Therefore, the atomic operations that manipulate the clock and the timestamp could have an impact here. We did batch the operations but we did not measure their impact. "Page activations" can occur at a faster pace than "page demands", especially when the workload fits in memory; but most importantly, "page activations" reveal that memory is being reused, i.e., that distinct accesses are occurring multiple times.

On the slowest time scale, we could also consider the event "hierarchically limited". When an internal cgroup reaches its limit, memory consolidation begins. But the full transfer is cut up in multiple rounds. The first round of page transfers could be wrong but given the "page demands" and "page activations" in between two "hierarchically limited" events, one

[4]The `try_charge` function captures page demands.
[5]The `__activate_page` function captures page activations.
[6]The `shrink_page_list` capture page activations in some cases.

could detect and correct the reclaim decisions. We did not use these events in our solution because we wanted to avoid the error of the first round.

The following subsections describe our event-based solutions: `dc`, `acdc`, and `sacdc` (later evaluated in Chapter 7).

`dc`: A solution based on "page demands" only

The acronym **`dc`** stands for "**d**emand **c**lock". This solution was published in NCA2017 [12] under the name of ACDC. But in this document, we decided to reuse the acronym ACDC to refer to another solution described below. `dc` defines the most recently active container as the one which has demanded a page the most recently.

To implement `dc`, we added *i)* a `global_clock` which is atomically incremented every time a cgroup demands a new page[4] and *ii)* a `demand_clock` per cgroup, i.e., a timestamp that stores the last time the cgroup demanded a page. `dc`'s policy is characterized by the following functions:

- `reclaim` returns the current consumption of the cgroup.
- `cmp` compares the `demand_clocks` of the cgroups.
- `scan` returns 0.

However, page demands are not enough, because containers that recently demanded a page might possess unused pages worth evicting. Moreover, if an active cgroup fits its data in memory and stops asking for memory, it would then be falsely detected as inactive. Nevertheless, `dc` provides a good reactivity because if pages are mistakenly taken from this cgroup, as soon as it asks them back it will be protected.

`acdc`: adding "page activations" to the scheme

The acronym **`acdc`** stands for "**a**ctivation **c**lock and **d**emand **c**lock" and attempts to incorporate "page activations" in the decisions of `dc`.

To implement `acdc`, we added a `activate_clock` per cgroup which is updated every time the cgroup activates N pages according to the formula that follows:

$$\texttt{activate_clock}_{n+1} = max(\texttt{activate_clock}_n, \texttt{global_clock}) + N \tag{6.1}$$

Thus, when a cgroup activates N pages, its `activate_clock` is at most N times further in the future (given that the present is the `global_clock`). Then, we characterized `acdc`'s policy by the following functions:

- `reclaim` returns the current consumption of the cgroup.

- `cmp` compares the max(`activate_clock`,`demand_clock`) of the cgroups.
- `scan` returns 0.

In short, if we define max(`activate_clock`,`demand_clock`) as the `age` of a container, then `acdc` considers that the most recently active container is the youngest container. Thus, if containers never activate their pages, their `age` is equal to their `demand_clock` and `acdc` behaves exactly like `dc`. But most importantly, when containers do activate their pages, `acdc` offers them an extra window of protection against containers that never activate their pages.

`sacdc`: triggering "page activations" with extra `force_scan`s

The acronym **`sacdc`** stands for "**s**can protected, **a**ctivation **c**lock and **d**emand **c**lock" and attempts to keep the age of the youngest containers synchronized. `dc` and `acdc` do not touch the lru of the youngest containers when memory is reclaimed in the oldest containers. Thus, if an active cgroup fits its data in memory, its `activate_clock` and `demand_clock` will lag behind the `global_clock` when an inactive cgroup activates itself and reclaims memory in the oldest containers.

To implement `sacdc`, we characterized its policy by the following functions:

- `reclaim` returns the current consumption of the cgroup.
- `cmp` compares the `age` of the cgroups.
- `scan` returns the current consumption of the cgroup.

In short, `sacdc` refreshes the lru of the youngest containers and gives them the opportunity to stay young by triggering more activation after memory has been reclaimed in the oldest containers.

6.4 Conclusion

This chapter summarized our modifications to the Linux kernel. We first developed `force_scan`, a mechanism that executes the PFRA without reclaiming the pages. In a second step, we reordered the loop of the PFRA over cgroups to the outmost position. This reordering allowed us to enforce `soft_limits` at any level of the hierarchy. In a third part, we added an extra loop on top of the PFRA to stack generic policies which are applied from the least strict to the strictest. Finally, we implement new policies such as `dc`, `acdc`, and `sacdc` which take decisions based on kernel events, but also `reclaim_order` that lets userspace take the decisions from metrics or perfect knowledge of the workloads.

CHAPTER 7

EVALUATION OF THE METRIC AND THE EVENT-DRIVEN APPROACHES

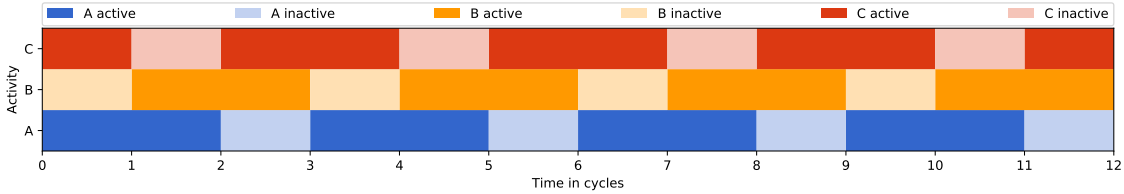
This chapter presents the evaluation of the different solutions presented in Chapter 6. The methodology used in this evaluation is more complex than the one used in the experiments of Chapter 4. The solutions are compared against each other in terms of performance and consolidation errors, with respect to the least performing server. Moreover, these results are also put into perspective with baseline and control experiments.

7.1 Experimental setup

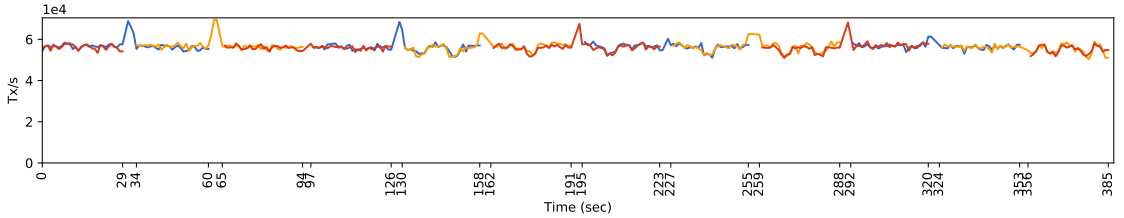
We first wanted to directly evaluate the solutions using the workload in production at Magency [73]. The plan was to use goreplay [83], a man-in-the-middle utility, to record and redirect the traffic in production to a duplicate machine where our kernel would be deployed. But this plan was abandoned because it would have required the development of a middleware to rewrite tokens on the fly [80]. Consequently, we developed another methodology by modeling the activity shifts of workloads in production which we deployed at the scale of the machine depicted in Section 4.2.1.

7.1.1 Workload's types and inactivity models

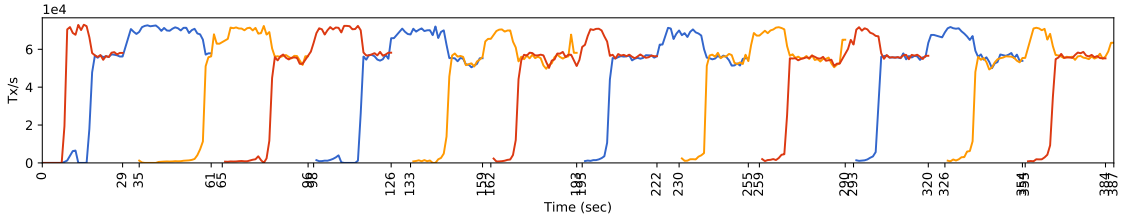
The inactivity model introduced in the experiment of Section 4.2.1 is not enough to compare the solutions because *i)* the inactive container is comparable to a frozen container (recall cgroup freezer in Section 2.2.1), and *ii)* the delay between the deactivation and the activation of containers is not as short as possible. We need some activity in the inactive container to justify not having to freeze it and we need the aforementioned delay to be as



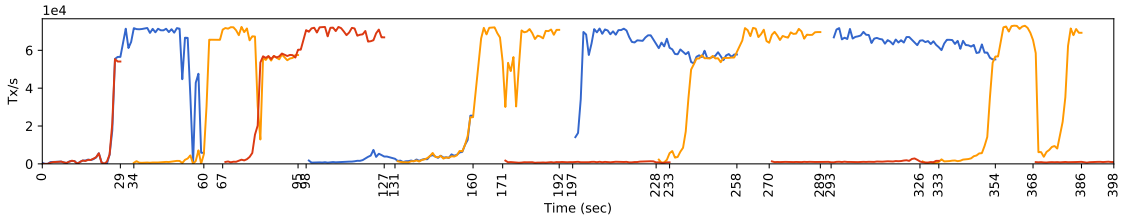
(a) A, B and C are active two thirds of the time.



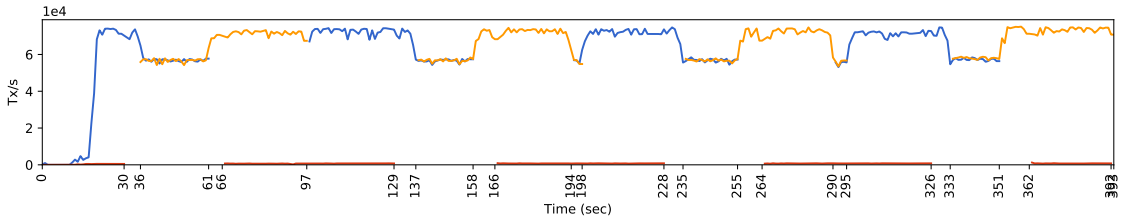
(b) Infinite amount of memory (`inf`).



(c) Without consolidation errors (`opt`).



(d) With consolidation errors (`nop`).



(e) Without fairness (`unf`).

Figure 7.1: Transaction rate of active Redis servers under the low request rate inactivity model.

short as to possible to highlight the most reactive solutions. Moreover, the experiment of Section 4.2.1 only uses "out of memory" workloads but we also need to study "in memory" workloads[1] because their inactivities are harder to detect (recall Chapter 5).

A total of 4 cases are used to evaluate the metric-based solutions and the event-based solutions. These cases are built from two types of workloads with two types of inactivity model. The "out of memory" workload type is represented by MySQL servers that receive requests from Sysbench clients [144] and the "in memory" workload type is represented by Redis servers that receive requests from Memtier clients [74]. To simulate low phases of activity in the first model, 10% of the network packets of the inactive client are dropped to decrease the incoming request rate. In contrast, the second model does not throttle the request rate but configures them to hit only 1% of the whole dataset to decrease the server's workingset size.

7.1.2 Schedule of activity shifts and configuration of resources

The machine used for the evaluation is the same as the one depicted in Section 4.2.1 but the scenario of the experiment is different. All three servers are homogeneous in size and type, i.e., there are either three MySQL servers or three Redis servers. The experiments last about 6 minutes and during that time frame, every server becomes inactive 4 times out of 12, i.e., every 30 seconds the single inactive server becomes active and the server that was active for the longest time becomes inactive (see Figure 7.1a). Therefore, there are always only two servers active simultaneously. The servers are deployed on three isolated cores using the cgroup cpuset and the fourth core of the machine is shared by the client request generators. Additional monitoring tools required by the metric-based solution are allowed to consume CPU cycles anywhere but their cgroup cpushares are set to the minimum.

7.1.3 Throttling issues with Blkio cgroup

In Section 2.4.4, we showed that CFQ can provide isolation and consolidation of disk bandwidth at the root of the cgroup hierarchy, but in the evaluation we faced the need to ensure this property at the level of internal nodes. Indeed, the SSD at disposal is too fast to unveil consolidation errors. As explained in Section 4.2.3, if the active server has enough bandwidth to the disk, its performance will not be impacted by the pages lost due to consolidation errors. Ideally, we would like to dedicate most of the disk bandwidth to the activating server[2] and keep the consumption of the other servers relatively low because *i)* they are either inactive and therefore do not have to reload the page they are losing, or *ii)* they are either active and therefore should not lose any pages during consolidation. Unfortunately, the blkio cgroup does not support hierarchical throttling, i.e., child cgroups do not respect the limit of their parents. We, therefore, decided to deploy the servers in the same blkio cgroup to limit their total disk bandwidth. As the servers are homogeneous, we can tolerate some competition on the disk bandwidth between the processes of the servers.

[1] Workloads which fit entirely in memory.

[2] The activating server is the one which is waking up.

	Configuration	Total Memory	Transfers Memory	Updates `lru`/metric	Comment
Baseline	`nop`	2	Yes	No	≈ Linux 4.6.0
	`opt`	2	Yes	No	Best
	`unf`	2	No	No	Worst
Event	`dc`	2	Yes	No	demands
	`acdc`	2	Yes	No	+activations
	`sacdc`	2	Yes	Lazily	+`force_scans`
Metric	`rr1%`	2	Yes	No	CPU usage < 1%
	`rrs`	2	Yes	Yes	cpushare = 2
	`rrs10%`	2	Yes	Yes	CPU usage < 10%
	`ir`	2	Yes	Yes	cpushare = 2
	`ir10%`	2	Yes	Yes	CPU usage < 10%
Control	`inf`	3	No	No	Wastes memory
	`irrs`	3	No	Yes	cpushare = 2
	`irrs10%`	3	No	Yes	CPU usage < 10%
	`iir`	3	No	Yes	cpushare = 2
	`iir10%`	3	No	Yes	CPU usage < 10%

Table 7.1: Experimental configurations.

7.1.4 Experimental configurations

Before interpreting the data of the evaluation, we first have to introduce the different configurations and show some experiment examples that have not been aggregated over time (see Figure 7.1). Table 7.1 summarizes the experimental configurations described in the following subsections. The Table also includes the metric and the event-based configurations that have been described in Chapter 6.

Control configurations

The acronym **`inf`** stands for "infinite amount of memory". In this configuration, memory is not consolidated, i.e., A, B and C have enough memory to run simultaneously but since they do not, memory is wasted. Figure 7.1b shows that once data has been loaded in memory, there is no warming up phase when a server awakes.

The acronym **`iir*`** (resp. **`irrs*`**) does not consolidate memory but unlike `inf`, it additionally tries to update the *idle ratio* (resp. the *rotate ratio*) metric by using the same throttling option as `ir*` (resp. `rrs*`). The goal is to show interferences on workloads due to metric computation (recall Section 6.3.1).

Baseline configurations

The acronym **`nop`** stands for "no operation" and is the closest configuration to Vanilla Linux 4.6.0. As rebooting to switch kernel between experiments takes time, we decided to merge

inf	irrs10%	irrs	iir10%	iir	opt	dc	acdc	sacdc	rr1%	rrs10%	rrs	ir10%	ir	nop
171939	165707	160143	171526	163502	160148	159842	158260	156217	121017	133114	133933	156280	150782	121667
107.4%	103.5%	100.0%	107.1%	102.1%	100.0%	99.8%	98.8%	97.5%	75.6%	83.1%	83.6%	97.6%	94.2%	76.0%

(a) MySQL under the low request rate model.

inf	irrs10%	irrs	iir10%	iir	opt	dc	acdc	sacdc	rr1%	rrs10%	rrs	ir10%	ir	nop
145259	136876	131190	138565	133562	136989	121702	122892	115320	117780	107947	101406	134240	122575	115532
106.0%	99.9%	95.8%	101.1%	97.5%	100.0%	88.8%	89.7%	84.2%	86.0%	78.8%	74.0%	98.0%	89.5%	84.3%

(b) MySQL under the small workingset size model.

inf	irrs10%	irrs	iir10%	iir	opt	dc	acdc	sacdc	rr1%	rrs10%	rrs	ir10%	ir	nop
13539348	13448924	12865025	13494763	13002547	10108396	5341220	4944667	5546386	249276	201459	1007273	5357629	3690111	1146460
133.9%	133.0%	127.3%	133.5%	128.6%	100.0%	52.8%	48.9%	54.9%	2.5%	2.0%	10.0%	53.0%	36.5%	11.3%

(c) Redis under the low request rate model.

inf	irrs10%	irrs	iir10%	iir	opt	dc	acdc	sacdc	rr1%	rrs10%	rrs	ir10%	ir	nop
11121394	10677308	9634332	10788554	9874143	8174636	5666684	5629504	6417177	252993	207185	227143	3532927	1927761	329864
136.0%	130.6%	117.9%	132.0%	120.8%	100.0%	69.3%	68.9%	78.5%	3.1%	2.5%	2.8%	43.2%	23.6%	4.0%

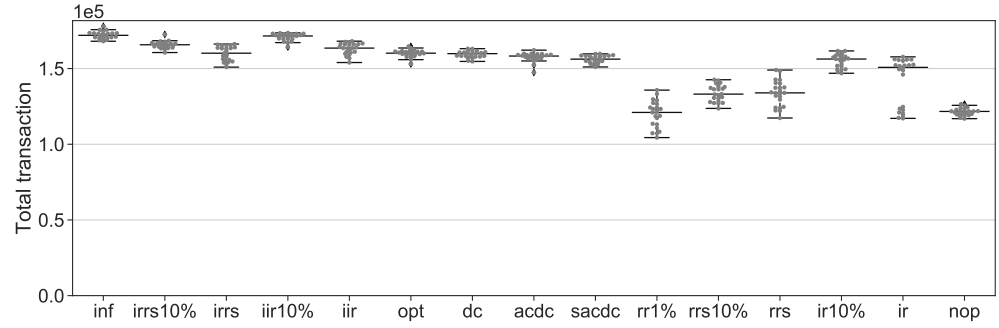
(d) Redis under the small workingset size model.

Table 7.2: Medians of values presented in Figure 7.2 relative to `opt`.

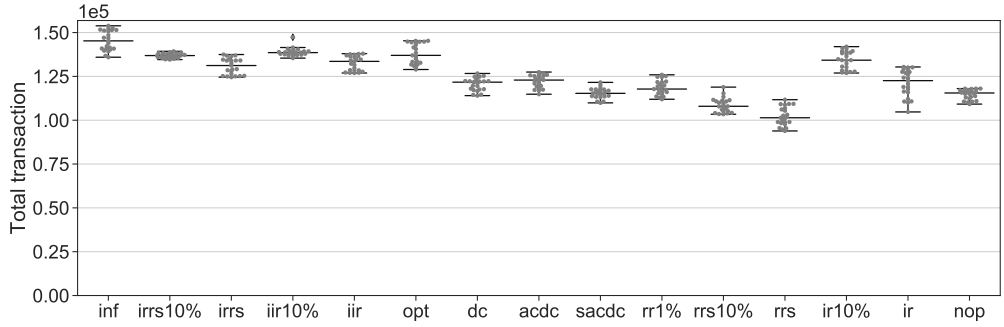
all mechanisms into one kernel and enable or disable them when needed. Therefore, `nop` does update the event clocks[3] but do not use their values because it does not order containers during memory reclaims. Figure 7.1d shows the consolidation errors of `nop`. For example, B is active at time 386 seconds, but its transaction rate drops when C wakes up. Moreover, waking up containers do not always reach the maximum transaction rate (for examples: A at time 98 or C at time 171).

The acronym **`opt`** stands for "optimal" and is the closest configurations to the ideal solution, the one that perfectly guesses which containers are active and which are inactive (recall Section 6.2). As there is always only one server inactive in the schedule, we decided to keep the scheduling script simple and use the implementation of `opt` based on `soft_limits`, but if more than one server was inactive, we would have to use an implementation based on `reclaim_orders` (recall Section 6.2.1). Figure 7.1c shows that consolidation errors can be avoided if the user can correctly provide soft limits that match activity shifts.

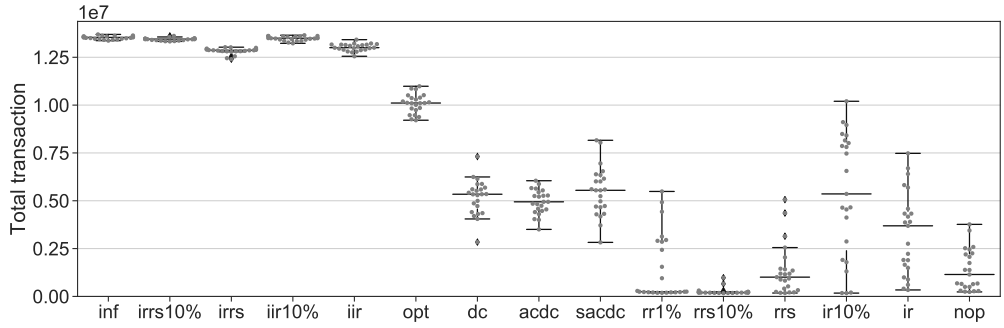
The acronym **`unf`** stands for "unfairness". In this configuration, memory is unfairly always reclaimed to C to protect the memory of A and B. To implement `unf`, we simply set the `soft_limit` of C to 0 and those of A and B to the maximum. `unf` could outperform `opt`, but in our experiments, it does not because the time required to swap the data of a container in and out of memory is short enough compared to the time containers stay inactive. Figure 7.1e shows that A and B do not have to reload their data from disk at the cost of wasting memory and providing a highly degraded service in C.



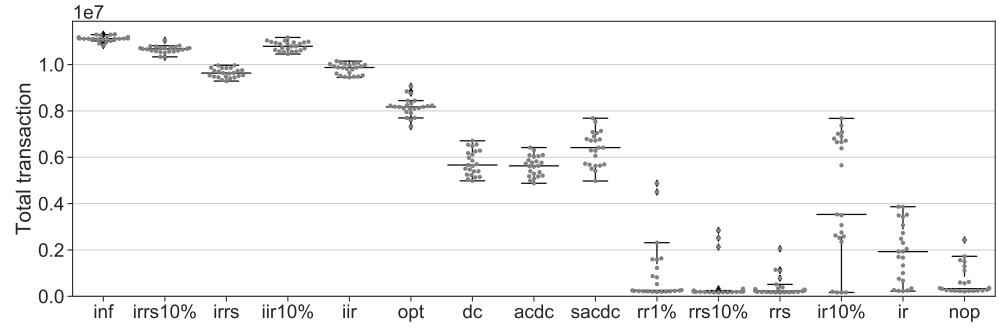
(a) MySQL under the low request rate model.



(b) MySQL under the small workingset size model.



(c) Redis under the low request rate model.



(d) Redis under the small workingset size model.

Figure 7.2: Transactions of the least performing active server.

7.2 Performance analysis

We first chose to compare the different configurations with respect to the total number of transactions carried out by all the servers over the time that they were active. Indeed, we chose to ignore the requests issued during the low phases of activity because they are either very scarce under the low request rate model or too easy to serve under the small workingset size model. Unfortunately, despite working properly with MySQL servers, this first comparing method did not provide clear insights in the case of Redis servers. Indeed, in that case, it turns out that `unf` outperforms all solutions (except `opt`) in terms of total transaction carried out by active servers. Consequently, with this comparing method, a solution can "cheat" and outperform other solutions by simply behaving like `unf` instead of trying to behave like `opt` (see Figures 7.1d and 7.1e: `nop` behaves like `unf` between time 258 and 333 seconds). Therefore, to reject this behavior, we decided to compare solutions with respect to the total number of transactions carried out by the least performing server[4] over the time that it was active. Fortunately, the results obtained with MySQL servers are very similar with both methods.

The experiments have been repeated at least 20 times and the results have been reported in Figure 7.2 which plots their median, quartiles, and distributions. The medians and their value relative to `opt` are reported in Table 7.2.

7.2.1 Control Experiments

The control experiments show that the metric computation has an impact on the workload (`inf` outperforms all configurations). As expected, this effect grows when the metric is updated more frequently (`iir10%` outperforms `iir` and `irrs10%` outperforms `irrs`), but to the point where it can outweigh the benefit of correctly consolidating memory (`opt` outperforms `iir` and `irrs` in Figure 7.2b). The control experiments also reveal that given the same CPU restrictions, the *rotate ratio* metric computation has a higher impact on the workload than the *idle ratio* computation (`iir` outperforms `irrs` and `iir10%` outperforms `irrs10%`).

7.2.2 Event-based solutions

The event-based solutions are quite good complements to each other. In the case of MySQL under the low request rate model (see Figure 7.2a) `dc` outperforms all the other solutions (including the metric-based). `acdc` results are very close to `dc`: slightly worst in two cases (see Figures 7.2a and 7.2c), almost identical in one case (see Figure 7.2d) but slightly better in one case (see Figure 7.2b). However, `sacdc` is a good improvement over `dc` and `acdc` in the case of Redis servers (see Figures 7.2c and 7.2d). Unfortunately, it is less efficient in the case of MySQL servers (see Figure 7.2a) and it could even be considered as a regression to

[3] All configurations update the clocks but the overhead is too small to compromise the results.

[4] The least performing server is the server that carried out the least number of transactions.

`nop` (see Figure 7.2b). Even if there is no best solution across all workloads, `dc` seems to always be a good improvement over `nop`.

7.2.3 Metric-based solutions

The *rotate ratio* was the very first metric we studied to guide our solution. Unfortunately, as the evaluation suggests, the *rotate ratio* is not a good candidate. Even if `rrs10%` and `rrs` are improvements in the case of MySQL under the low request rate model (see Figure 7.2a), they can neither handle it under the small workingset model (see Figure 7.2b) nor can they handle Redis cases (see Figures 7.2c and 7.2d). Surprisingly, `rr1%` did not perform poorly everywhere; in the case of MySQL under small workingset model (see Figure 7.2b), it was able to do slightly better than `nop`; we will try to explain this exceptional result in the next section.

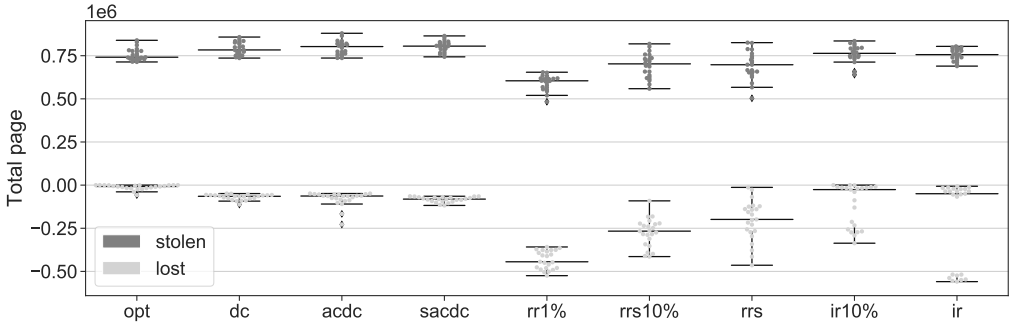
The *idle ratio* metric is a very good candidate. It almost performs as well as `dc` in the case of MySQL under the low request rate model (see Figure 7.2a) and provides outstanding results under the small workingset model (see Figure 7.2b). Unfortunately, in the case of Redis, the results of the *idle ratio* metric are too spread apart for it to be considered better than the event-based solutions. Moreover, the assumption that all idle CPU should be invested into updating the metric more frequently does not hold in the case of the *idle ratio* because `ir10%` always outperforms `ir`.

7.3 Page transfer analysis

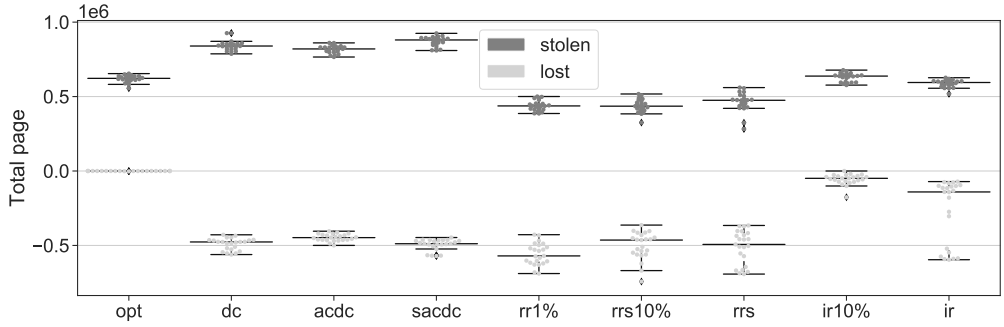
The problem at its core is not about performance but about correctly transferring memory from inactive containers to active containers. Logging every page transfer would have been an overkill process to measure the correctness of a solution. Therefore, in Chapter 4, we introduced the `pgstolen` and `pglost` counters to measure the memory transfers. Even if these counters neither keep the identity of the cgroup from which memory was stolen nor the identity of the cgroup to which memory was lost; they provide enough information to highlight the strengths and weaknesses of the different solutions.

We decided to study the total page transferred in (`pgstolen`) and out (`pglost`) of the least performing server over the time that it was active (see Figure 7.3). Other kinds of visualizations are possible but we decided to stay coherent with the view chosen in the previous section on performance analysis.

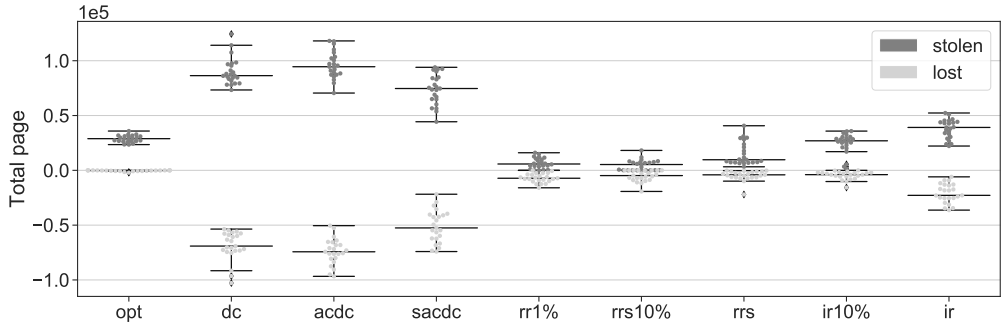
Since the server is considered to be active, it should not lose any pages. As expected, in all Figures, `pglost` values are almost always equal to zero with `opt`. Moreover, the active server should be able to grow to its full capacity by stealing pages. A theoretical amount of total page stolen could be deduced from the memory limit and the number of active phases but in practice, inactive cgroups are not shrunk down to zero pages and active cgroups end up recycling some of their own pages. Thus, we can interpret that the server should at least be able to steal just as many pages than it would have been able to steal under the `opt` solution.



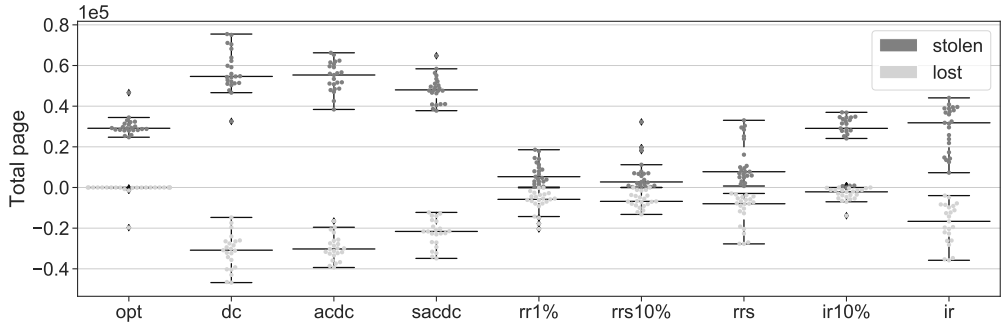
(a) MySQL under the low request rate model.



(b) MySQL under the small workingset size model.



(c) Redis under the low request rate model.



(d) Redis under the small workingset size model.

Figure 7.3: Page transferred in and out of the least performing active server.

7.3.1 *Rotate ratio* solutions

Given the aforementioned guidelines, we can deduce that the major flaw of the *rotate ratio* solutions is that they do not allow the server to grow to its full capacity. Indeed, the `pgstolen` values of `rr1%`, `rrs10%`, and `rrs` are significantly lower than `opt` compared to that of the other solutions (see Figures 7.3b, 7.3c and 7.3d).

The assumption that all idle CPU should be invested into updating the metric more frequently does hold for the *rotate ratio* in Figure 7.3a. Indeed, even if there is no difference in the number of pages stolen between `rrs10%` and `rrs`, `rrs` does reduce the number of pages lost. But this improvement is not significant enough compared the cost of updating frequently the metric because `rrs` does not outperform `rrs10%` (recall Figure 7.2a).

Surprisingly, in Figure 7.3b, we can observe that the server loses more pages than it steals, especially in the case of `rr1%`. The explanation behind these results is that under this particular workload, the metric wrongly evaluates the inactive server as being active. Thus, when the server really becomes active, it is allowed to grow until its *rotate ratio* slowly grows back to the point where it becomes detected as inactive, despite being still active under our definition. This small window of memory protection at the beginning of the active phase is enough to give the server the opportunity to outperform the results of `nop` (recall Figure 7.2b).

7.3.2 *Idle ratio* solutions

Compared to all the other solutions, we can observe in Figures 7.3a to 7.3d that `ir10%` is the solution which has the closest values to that of `opt`, both in terms of `pglost` and `pgstolen`. However, the tight spread of `ir10%`, observed in Figures 7.3c and 7.3d does not explain its wide dispersion in the performance results observed in Figures 7.2c and 7.2d. This difference suggests that `ir10%` is taking the same decisions than `opt` but that there is a random delay between the activity shift and the detection of that shift by the metric.

Case-by-case analysis revealed that the *idle ratio* of the least performing server sometimes struggles to cross over the *idle ratio* of the recently inactive server. It sometimes takes a long time to cross over and sometimes does not cross at all. Therefore, `ir10%` sometimes behaves, in the best cases, like `opt`; sometimes behaves, in the worst cases, like `unf`; and sometimes behaves like an `opt` with delayed decisions. Nevertheless, our data show that these errors are not due to an insufficient amount of CPU dedicated to the computation of the metric. Indeed, `ir` deteriorates the quality of the decisions taken by `ir10%`: with MySQL, there are more `pglost` (see Figures 7.3a and 7.3b) and with Redis, there are more `pglost` and `pgstolen` (see Figures 7.3c and 7.3d). The *idle ratio* metric might just have hit the limitation of the accessed bit as described by N. Agarwal [2].

7.3.3 Event-based solutions

In Chapter 5, we have shown the CPU cost of metric-based solutions but nothing related to event-based solutions. However, just like `ir10%`, event-based solutions are not free. They

introduced another kind of cost, distinguishable from the CPU cost of metric computation, that can be measured against `opt`. In Figures 7.3a to 7.3d, we can observe that `dc`, `acdc`, and `sacdc` have significantly higher values than `opt` and `ir10%`, both in terms of `pglost` and `pgstolen`. Indeed, event-based solutions learn from their mistakes. They do a lot of `pglost`, which are still consolidation errors, but they compensate these errors by letting the server steal more pages. In other words, even if `dc`, `acdc`, and `sacdc` are huge improvements over `nop`, they do not eradicate the extra amount of `pglost` and thus, still exploit extra disk bandwidth to reload data.

In Figures 7.3c and 7.3d, we can observe that even if `sacdc` significantly reduces the `pglost` of `dc` and `acdc`, `sacdc` fails to reach the values of `opt`, unlike `dc` in Figure 7.3a. Moreover, there is a lot of room for improvement in Figure 7.3b. Indeed, `acdc` barely reduces the `pglost` of `dc`. However, the fact that `ir10%` is able to consistently handle this case suggests that it would not be hard to devise a new event-based mechanism to handle MySQL under the small workingset inactivity model.

7.4 Conclusion

The scenario designed in Section 4.2.1 was not intensive enough to challenge our solutions. In this chapter, we design four scenarios to highlight the best solutions. Even if there is no best solution across all scenarios, `dc` seems to always be a good improvement over Linux 4.6.0. The control experiments show that the cost of CPU time dedicated to monitoring the metrics can outweigh the benefit of correctly consolidating memory. In terms of `pglost` and `pgstolen`, `ir10%` has the closest values to that of `opt`, but its decisions still lag behind that of `opt`. Even if the event-based solutions are more reactive, they do some consolidation errors (`pglost`) which they have to compensate (with more `pgstolen`). The *rotate ratio* solutions performed poorly but it should not be discarded because the evaluation did not consider workloads where they could shine. The *rotate ratio* solutions could outperform the others on a very specific type of workloads consuming a lot of pages that are used once and never used again: these include database scans, dumps or bulky insertions.

CHAPTER 8

CONCLUSION AND FUTURE WORKS

The Cloud has made computing infrastructures accessible to everyone by relieving the burden of building and maintaining data centers. Cloud providers have realized economies of scale by pooling our resource needs under one roof, but the next step towards a Cloud without waste is to oversell virtual resources. In contrast to other resources such as CPU time or disk bandwidth, memory is harder to consolidate and it has been shown that virtual machines are not flexible enough to tackle this challenge. Containers, on the other hand, are a promising technology that could minimize the waste of resources. However, this thesis shows that Linux containers cannot consolidate memory without disturbing the performance of the most active containers. Indeed, consolidation must not reclaim useful pages of the most active containers if there are unused pages in the most inactive containers. An analysis of the kernel code reveals that isolation has to split the memory pools, but without a global pool, the least recently used page of the machine cannot be tracked. Our solutions propose that consolidation must target inactive containers first to protect active containers. Two approaches are suggested to build a relative order of the memory needs between containers. The first idea considers existing memory-related metrics such as the *rotate ratio* and the *idle ratio* to rate containers, while the second idea considers existing memory-related events such as page demands and page activations.

The goal of this thesis is to preserve isolation during resource consolidation. This resource property should hold for memory as it is already the case for other resources. Indeed, in Chapter 2, we illustrate that containers are able to consolidate CPU time and disk bandwidth while preserving performance isolation. However, Chapter 3 explains that the property cannot hold for memory because pages are reclaimed from any cgroups. Thus, in Chapter 4, we demonstrate that when Linux has to reclaim memory for a booting Cassandra server, it does not make the distinction between a MySQL server that receives a thousand requests per second from another MySQL server that did not receive any requests. In Chap-

ter 5, we evaluate the relevance of the metrics selected to detect activity patterns. We show that the *rotate ratio* takes a few seconds to detect that `anon` pages are less useful than `file` pages. But in most cases, this metric requires additional scans to stay up to date. We show for example that it takes on average 409 microseconds to scan 256 pages per second. The evaluation of the *idle ratio* shows that the memory needs of containers can be correctly estimated. However, the metric output is not continuous and has to be computed for all containers on the machine. Given 10% of CPU time, it takes 35 seconds to detect if 800 MB out of 1 GB was accessed or not.

Chapter 6 presents our solutions that are classified into two approaches. On the one hand, the metric-driven approach: `ir` which uses the *idle ratio*, `rr` which uses the *rotate ratio* and `rrs` which uses `force_scan`. On the other hand, the event-driven approach: `dc` which uses "page demands", `acdc` which uses "page activations" and `sacdc` which uses `force_scan`. In Chapter 7, we show, in the case of MySQL workloads, that it is possible to achieve performance results close to an optimal solution that perfectly adjusts the memory needs order when the activity shifts. The median value of `dc` reaches 99.8% of the median value of `opt` under the low request rate model. The median value of `ir10%` reaches 98.0% of the median value of `opt` under the small workingset size model. However, the results are less impressive in the case of Redis workloads: the median value of `sacdc` reaches only 54.9% of the median value of `opt` under the low request rate model, and 78.5% of the median value of `opt` under the small workingset size model. The metric-based solutions produced results that are too scattered compared to the event-based ones. The page transfer analysis suggests that `ir10%` transfers memory like `opt` but with a slight delay. It also suggests that event-based solutions make more transfers than necessary, but overall the active server performs well because it is allowed to steal more pages than the amount it lost.

Our final recommendation is `dc`, because it is always a good improvement over Linux 4.6.0, even if there is no best solution across all scenarios. Nevertheless, `dc` has room for enhancement: compared to all the other solutions, `dc` never considers secondary accesses to data. Indeed, our attempts to incorporate this information in `dc` were not always fruitful. It would therefore be wise to suggest `dc` to the Linux kernel community for help and feedback.

8.1 Short-term challenges

Our prototypes show that it is feasible to consolidate memory without stealing pages to active containers when there are reclaimable pages in an inactive container. But they are two concerns to address before deploying them in production. First, the solutions could introduce contention on the most inactive containers; and second, the solutions make no improvement when all containers are active.

8.1.1 Spreading contention on the most inactive containers

Our global selective policy might introduce latencies during global memory reclaims because it creates contention on the most inactive containers. Contentions will occur if the number of pages reclaimed in the most inactive container is too small compared to the time spent waiting for acquiring the locks of the memory pools. This situation could be worst for example with write-intensive workloads because their pages would first have to be written back to disk before eviction. Moreover, as the most inactive container shrinks down, there is less job to parallelize because there are fewer pages to scan. Our solution could be relaxed to consider these contentions: if it detects contention on the most inactive container, the next memory reclaim could skip it and directly start on the second most inactive container.

8.1.2 Ensuring properties when all containers are active

By now, our solutions offer no improvement when all containers are active because we first had to deal with the case where there is always a most inactive container to take memory from. Thus, the experiments presented in Section 2.4 still cannot be reproduced for the case of memory (recall that in these experiments, all containers are active at time 20 and that consolidation is required at time 60). Indeed, before this thesis, there were two options available. On the one hand, Linux could always ensure isolation but without consolidation, i.e., by setting the sum of the `hard_limit` of the children to a value lesser than or equal to that of their parent. On the other hand, Linux could try to consolidate memory (with mechanisms such as `soft`, `min` and `max limits`) but without isolation guarantees, as we have shown in this thesis. Today, thanks to our work, the second option was enhanced. Linux can consolidate memory with isolation guarantees for active containers when some of them are inactive. However, if the `hard_limits` are set up to consolidate memory, Linux will still be unable to ensure isolation properties when all containers are active. Thus, in this particular case, the state of the art remains unchanged: all or any containers could be thrashing and this undefined behavior has to be specified.

One way to improve our solutions would be to provide heuristics which detect when a container is active or inactive on an absolute basis, i.e., independently of the current memory consumption of the container and independently of the workloads running in the other containers. Such heuristics could be devised by comparing a user-defined threshold to the speed of distinct memory accesses normalized with respect to the current memory consumption. Memory would first be reclaimed to the class of inactive containers using one of the approaches proposed in this thesis. Then, if the class of inactive containers was not able to satisfy the memory shortage, the solution would be able to notify the event “All containers are active” and trigger the desired policy in that case.

Many policies could be conceived when all containers are active. First, we could decide to freeze the current partitioning state by forbidding containers to steal memory to their active siblings. This policy would force containers to reclaim their own memory and upon failure, trigger the OOM killer on themselves. Second, we could add priorities and decide

that some active containers are more important than others. For instance, interactive containers which respond to external requests could be allowed to steal memory from active containers processing batched jobs that can be delayed. An interactive container would only give back the memory when its heuristic detects it as inactive. Third, we could decide to enforce fairness between active containers by generalizing the swap token algorithm at the scale of containers. Just like a scheduler, its goal would be to fairly elect a single active container that would be the only one able to steal memory from its active siblings during a given time window of protection.

Thus, by introducing the "All containers are active" event in the scheme, memory could achieve the properties illustrated in Section 2.4 for CPU and disk bandwidth. But it is unclear if it is always possible to detect such an event because containers with little memory will struggle to appear as active as containers with more memory. Ideally, it would be wise to prove that it is impossible to always ensure both isolation and consolidation without detecting the "All containers are active" event.

8.2 Long-term perspectives

Once Linux is able to protect active containers from inactive ones, Cloud providers will be able to stack inactive containers on a single machine until they reach the point where the amount of resources needed by active containers overwhelms the capacity of the machine. Beyond that point, two options prevail. On the one hand, multiple machines can be used to increase the capacity and meet the need of active containers. This option will have to extend the isolation and consolidation properties to the scale of a cluster. On the other hand, as a single machine cannot meet the need of all active containers, a Pareto optimum can be used to maximize global performance. This option will have to study how performance relates to resource allocation.

8.2.1 Ensuring isolation and consolidation at the scale of a cluster

Our work can be part of a broader perspective. We consider integrating these mechanisms at the level of a cluster of machines through orchestrators such as Kubernetes [71] or Mesos [54]. In this environment, in addition to the problem of resizing containers, the orchestrator can also migrate containers between nodes, boot up or shut down machines to save energy or money. We believe that combining our metric and event-driven approaches could help unlock some of these challenges. On the one hand, at the scale of the single node, the event base solutions are well suited to consolidate containers that quickly shift their activity pattern. On the other hand, absolute metrics such as the *idle ratio* are well suited to compare containers across multiple nodes. As machine boots and shutdowns take time, orchestrators must accurately predict activity patterns at this scale. These kinds of patterns are generally steady and therefore do not need metrics to be collected at a fast pace. Moreover, to further reduce the metric computation cost, the orchestrator could only monitor certain containers: those suspected of becoming active and those suspected of becoming inactive. Finally, if we are able to notify that all containers are active, these events

could serve as an incentive not only for booting or shutting down nodes when they are occurring at an extremely high or low pace; but also for migration when there is an imbalance in the rate at which they are occurring between nodes.

8.2.2 Maximizing global performance with limited memory

Applications use memory in various fashions, which makes it hard to guess their performance when some of their data are not present in memory. The most predictable applications will store their data on disk in the order of the access pattern. These applications will lightly suffer from missing data because when the very first accesses are issued, the prefetching will also bring the following data in memory. In contrast, applications that randomly access data on disk cannot take advantage of the sequential prefetching. Nevertheless, if they can predict their accesses at runtime, they can asynchronously request data missing in memory to avoid being blocked in the future.

In the literature, the Miss Ratio Curve (MRC) has been used to predict how memory consumption and application performance are related. Thus, in parallel of the thesis, I have collaborated with an intern, I. Toumlilt, and later on with an engineer, M. Bittan, to work on Linux refault distance. M. Bittan improved the precision of the refault distance. His work allows the kernel to estimate the MRC for memory values greater than the current consumption. However, estimating the MRC for values lower than the current consumption is challenging because the online measurement must not alter the workload performance. We suggested the use of `Cleancache` API to build an additional "dead pool" of pages that would grow and equally shrink the `lru`. Thus, accesses to pages in the "dead pool" would be monitored as misses, but the accesses would not be as expensive as real misses because the pages are not out of memory. This scheme faithfully emulates the exact behavior of pages in a smaller `lru`. However, it is unclear if the combination of a smaller `lru` with a "dead pool" would be equally able to emulate the exact evictions of the original `lru`.

Given the MRCs of the active containers, one could then decide performance trade-offs between containers. For instance, if containers have huge datasets of similar size which are randomly accessed with a uniform distribution, the Pareto front will simply be linear. However, for highly specified applications that require all their data to fit in memory, some of the active containers will have to be suspended. This case falls within the combinatorial optimization problem of knapsack.

BIBLIOGRAPHY

Academical references

[1] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, J. P. Shen and M. Martonosi, Eds. ACM, 2006, pp. 2–13. [Online]. Available: https://doi.org/10.1145/1168857.1168860

[2] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Y. Chen, O. Temam, and J. Carter, Eds. ACM, 2017, pp. 631–644. [Online]. Available: https://doi.org/10.1145/3037697.3037706

[3] N. Amit, D. Tsafrir, and A. Schuster, "Vswapper: a memory swapper for virtualized environments," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, R. Balasubramonian, A. Davis, and S. V. Adve, Eds. ACM, 2014, pp. 349–366. [Online]. Available: https://doi.org/10.1145/2541940.2541969

[4] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: a virtual mobile smartphone architecture," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, T. Wobber and P. Druschel, Eds. ACM, 2011, pp. 173–187. [Online]. Available: https://doi.org/10.1145/2043556.2043574

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

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[7] L. A. Belady, “A study of replacement algorithms for virtual-storage computer,” *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966. [Online]. Available: https://doi.org/10.1147/sj.52.0078

[8] L. A. Belady, R. A. Nelson, and G. S. Shedler, “An anomaly in space-time characteristics of certain programs running in a paging machine,” *Commun. ACM*, vol. 12, no. 6, pp. 349–353, 1969. [Online]. Available: https://doi.org/10.1145/363011.363155

[9] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. Van Doorn, J. Nakajima, A. Mallick, and E. Wahlig, “Utilizing iommus for virtualization in linux and xen,” in *OLS’06: The 2006 Ottawa Linux Symposium*. Citeseer, 2006, pp. 71–86.

[10] E. W. Biederman and L. Networx, “Multiple instances of the global linux namespaces,” in *Proceedings of the Linux Symposium*, vol. 1. Citeseer, 2006, pp. 101–112.

[11] A. Blin, “Vers une utilisation efficace des processeurs multi-coeurs dans des systèmes embarqués à criticités multiples. (towards an efficient use of multi-core processors in mixed criticality embedded systems),” Ph.D. dissertation, Pierre and Marie Curie University, Paris, France, 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01624259

[12] D. Carver, J. Sopena, and S. Monnet, “ACDC: advanced consolidation for dynamic containers,” in *16th IEEE International Symposium on Network Computing and Applications, NCA 2017, Cambridge, MA, USA, October 30 - November 1, 2017*, A. Gkoulalas-Divanis, M. P. Correia, and D. R. Avresky, Eds. IEEE Computer Society, 2017, pp. 253–260. [Online]. Available: https://doi.org/10.1109/NCA.2017.8171363

[13] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. D. Gligor, “Subdomain: Parsimonious server security,” in *LISA*, 2000, pp. 355–368. [Online]. Available: http://www.usenix.org/publications/library/proceedings/lisa2000/cowan.html

[14] P. J. Denning, “The working set model for program behavior,” *Communications of the ACM*, vol. 11, no. 5, pp. 323–333, 1968.

[15] Y. Dong, Z. Yu, and G. Rose, “SR-IOV networking in xen: Architecture, design and implementation,” in *First Workshop on I/O Virtualization, WIOV’08, San Diego, CA, USA, December 10-11, 2008, Proceedings*, M. Ben-Yehuda, A. L. Cox, and S. Rixner, Eds. USENIX Association, 2008. [Online]. Available: http://www.usenix.org/events/wiov08/tech/full_papers/dong/dong.pdf

[16] M. A. Eriksen, “Trickle: A userland bandwidth shaper for unix-like systems,” in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005, pp. 61–70. [Online]. Available: http://www.usenix.org/events/usenix05/tech/freenix/eriksen.html

[17] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*. IEEE Computer Society, 2015, pp. 171–172. [Online]. Available: https://doi.org/10.1109/ISPASS.2015.7095802

[18] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Badgertrap: a tool to instrument x86-64 TLB misses," *SIGARCH Computer Architecture News*, vol. 42, no. 2, pp. 20–23, 2014. [Online]. Available: https://doi.org/10.1145/2669594.2669599

[19] R. Ghosh and V. K. Naik, "Biting off safely more than you can chew: Predictive analytics for resource over-commit in iaas cloud," in *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*, R. Chang, Ed. IEEE Computer Society, 2012, pp. 25–32. [Online]. Available: https://doi.org/10.1109/CLOUD.2012.131

[20] R. P. Goldberg and R. Hassinger, "The double paging anomaly," in *American Federation of Information Processing Societies: 1974 National Computer Conference, 6-10 May 1974, Chicago, Illinois, USA*, ser. AFIPS Conference Proceedings, vol. 43. AFIPS Press, 1974, pp. 195–199. [Online]. Available: https://doi.org/10.1145/1500175.1500215

[21] S. E. Hallyn and A. G. Morgan, "Linux capabilities: Making them work," in *Linux Symposium*, vol. 8, 2008.

[22] S. Jiang, F. Chen, and X. Zhang, "Clock-pro: An effective improvement of the CLOCK replacement," in *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005, pp. 323–336. [Online]. Available: http://www.usenix.org/events/usenix05/tech/general/jiang.html

[23] S. Jiang and X. Zhang, "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2002, June 15-19, 2002, Marina Del Rey, California, USA*, R. R. Muntz, M. Martonosi, and E. de Souza e Silva, Eds. ACM, 2002, pp. 31–42. [Online]. Available: https://doi.org/10.1145/511334.511340

[24] ——, "Token-ordered LRU: an effective page replacement policy and its implementation in linux systems," *Perform. Eval.*, vol. 60, no. 1-4, pp. 5–29, 2005. [Online]. Available: https://doi.org/10.1016/j.peva.2004.10.002

[25] T. Johnson and D. E. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, J. B. Bocca, M. Jarke, and C. Zaniolo, Eds. Morgan Kaufmann, 1994, pp. 439–450. [Online]. Available: http://www.vldb.org/conf/1994/P439.PDF

[26] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018. [Online]. Available: http://arxiv.org/abs/1801.01203

[27] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[28] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *CoRR*, vol. abs/1801.01207, 2018. [Online]. Available: http://arxiv.org/abs/1801.01207

[29] M. Lorrillere, "Caches collaboratifs noyau adaptés aux environnements virtualisés. (A kernel cooperative cache for virtualized environments)," Ph.D. dissertation, Pierre and Marie Curie University, Paris, France, 2016. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01273367

[30] M. Lorrillere, J. Sopena, S. Monnet, and P. Sens, "Puma: pooling unused memory in virtual machines for I/O intensive applications," in *Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015, Haifa, Israel, May 26-28, 2015*, D. Naor, G. Heiser, and I. Keidar, Eds. ACM, 2015, pp. 1:1–1:11. [Online]. Available: https://doi.org/10.1145/2757667.2757669

[31] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*, C. Cole, Ed. USENIX, 2001, pp. 29–42. [Online]. Available: http://www.usenix.org/publications/library/proceedings/usenix01/freenix01/loscocco.html

[32] J. Lozi, B. Lepers, J. R. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The linux scheduler: a decade of wasted cores," in *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds. ACM, 2016, pp. 1:1–1:16. [Online]. Available: https://doi.org/10.1145/2901318.2901326

[33] P. Lu, Y. C. Lee, V. Gramoli, L. M. Leslie, and A. Y. Zomaya, "Local resource shaper for mapreduce," in *IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014, Singapore, December 15-18, 2014*. IEEE Computer Society, 2014, pp. 483–490. [Online]. Available: https://doi.org/10.1109/CloudCom.2014.55

[34] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 218–233. [Online]. Available: https://doi.org/10.1145/3132747.3132763

[35] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*. USENIX Association, 1993, pp. 259–270. [Online]. Available: https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet

[36] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[37] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. G. Shin *et al.*, "Performance evaluation of virtualization technologies for server consolidation," *HP Labs Tec. Report*, vol. 1, 2007.

[38] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," in *Proceedings of the Fourth Symposium on Operating System Principles, SOSP 1973, Thomas J. Watson, Research Center, Yorktown Heights, New York, USA, October 15-17, 1973*, H. Schorr, A. J. Perlis, P. Weiner, and W. D. Frazer, Eds. ACM, 1973, p. 121. [Online]. Available: https://doi.org/10.1145/800009.808061

[39] M. Quaritsch and T. Winkler, "Linux security modules enhancements: Module stacking framework and tcp state transition hooks for state-driven nids," *Secure Information and Communication*, vol. 7, pp. 7–13, 2004.

[40] N. Regola and J. Ducom, "Recommendations for virtualization technologies in high performance computing," in *Cloud Computing, Second International Conference, CloudCom 2010, November 30 - December 3, 2010, Indianapolis, Indiana, USA, Proceedings*. IEEE Computer Society, 2010, pp. 409–416. [Online]. Available: https://doi.org/10.1109/CloudCom.2010.71

[41] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008. [Online]. Available: https://doi.org/10.1145/1400097.1400108

[42] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975. [Online]. Available: https://doi.org/10.1109/PROC.1975.9939

[43] P. Sharma, L. Chaufournier, P. J. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*. ACM, 2016, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2988337

[44] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. C. Bavier, and L. L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March*

21-23, 2007, P. Ferreira, T. R. Gross, and L. Veiga, Eds. ACM, 2007, pp. 275–287. [Online]. Available: https://doi.org/10.1145/1272996.1273025

[45] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014.

[46] V. Tasoulas, H. Haugerud, and K. M. Begnum, "Bayllocator: A proactive system to predict server utilization and dynamically allocate memory resources using bayesian networks and ballooning," in *Strategies, Tools , and Techniques: Proceedings of the 26th Large Installation System Administration Conference, LISA 2012, San Diego, CA, USA, December 9-14, 2012*, C. Rowland, Ed. USENIX Association, 2012, pp. 111–121. [Online]. Available: https://www.usenix.org/conference/lisa12/technical-sessions/presentation/tasoulas

[47] B. Verghese, A. Gupta, and M. Rosenblum, "Performance isolation: Sharing and isolation in shared-memory multiprocessors," in *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 3-7, 1998.*, D. Bhandarkar and A. Agarwal, Eds. ACM Press, 1998, pp. 181–192. [Online]. Available: https://doi.org/10.1145/291069.291044

[48] G. Voron, G. Thomas, V. Quéma, and P. Sens, "An interface to implement NUMA policies in the xen hypervisor," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, G. Alonso, R. Bianchini, and M. Vukolic, Eds. ACM, 2017, pp. 453–467. [Online]. Available: https://doi.org/10.1145/3064176.3064196

[49] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, D. Boneh, Ed. USENIX, 2002, pp. 17–31. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec02/wright.html

[50] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013, Belfast, United Kingdom, February 27 - March 1, 2013*. IEEE Computer Society, 2013, pp. 233–240. [Online]. Available: https://doi.org/10.1109/PDP.2013.41

[51] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*. IEEE Computer Society, 2014, pp. 155–166. [Online]. Available: https://doi.org/10.1109/RTAS.2014.6925999

[52] Z. Zhuang, C. Tran, J. Weng, H. Ramachandra, and B. Sridharan, "Taming memory related performance pitfalls in linux cgroups," in *2017 International Conference on Computing, Networking and Communications, ICNC 2017, Silicon Valley, CA, USA, January 26-29, 2017*. IEEE Computer Society, 2017, pp. 531–535. [Online]. Available: https://doi.org/10.1109/ICCNC.2017.7876184

BIBLIOGRAPHY

Technical references

[53] "Amazon Web Services: on-demand cloud computing platforms." [Online]. Available: https://aws.amazon.com

[54] "Appache Mesos: program against your data center like it's a single pool of resources." [Online]. Available: http://mesos.apache.org/

[55] "AppArmor: an effective and easy-to-use Linux application security system." [Online]. Available: https://gitlab.com/apparmor/apparmor/

[56] "Association nationale de la recherche et de la technologie." [Online]. Available: http://www.anrt.asso.fr

[57] "AWS auto scaling: application scaling to optimize performance and costs." [Online]. Available: http://aws.amazon.com/autoscaling/

[58] "Cassandra: an open-source distributed storage system (docker image)." [Online]. Available: https://hub.docker.com/_/cassandra/

[59] "Cellrox: scalable, secured and robust mobile virtualization platform." [Online]. Available: http://www.cellrox.com/

[60] "Classful queuing disciplines." [Online]. Available: http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/classful-qdiscs.html

[61] "Classless queuing disciplines." [Online]. Available: http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/classless-qdiscs.html

[62] "CRIU: a project to implement checkpoint/restore functionality for Linux." [Online]. Available: https://www.criu.org

[63] "Docker: build, ship, and run any app, anywhere." [Online]. Available: https://www.docker.com/

[64] "Filebench Workload Model: describe desired workloads from scratch." [Online]. Available: https://github.com/filebench/filebench/wiki/Workload-model-language

[65] “Freezer-subsystem.” [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt

[66] “Google perfkit benchmarker: set of benchmarks to measure and compare cloud offerings.” [Online]. Available: https://github.com/GoogleCloudPlatform/PerfKitBenchmarker

[67] “Grsecurity: an extensive security enhancement to the Linux kernel.” [Online]. Available: https://grsecurity.net

[68] “High performance computing on AWS.” [Online]. Available: https://aws.amazon.com/hpc/

[69] “iCloud: the best place for all your photos, files, and more.” [Online]. Available: https://www.apple.com/icloud/

[70] “ioctl_ns: ioctl() operations for Linux namespaces.” [Online]. Available: http://man7.org/linux/man-pages/man2/ioctl_ns.2.html

[71] “Kubernetes: managing containerized applications across multiple hosts.” [Online]. Available: https://github.com/kubernetes/kubernetes

[72] “linux-ftools: Linux command line tools for fallocate, fincore, fadvise, etc.” [Online]. Available: https://code.google.com/archive/p/linux-ftools/

[73] “Magency: engaging and collaborative digital solutions for your business.” [Online]. Available: http://www.magency.me/

[74] “memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool.” [Online]. Available: https://github.com/RedisLabs/memtier_benchmark

[75] “MySQL: a widely used, open-source relational database management system (docker image).” [Online]. Available: https://hub.docker.com/_/mysql/

[76] “Overselling definition.” [Online]. Available: https://en.wikipedia.org/wiki/Overselling

[77] “proc: process information pseudo-filesystem.” [Online]. Available: http://man7.org/linux/man-pages/man5/proc.5.html

[78] “Senlin: auto scaling with Heat OpenStack.” [Online]. Available: https://docs.openstack.org/senlin/latest/scenarios/autoscaling_heat.html

[79] “Shadow: transform any device with a screen and internet connection into a high performance pc.” [Online]. Available: https://shadow.tech/

[80] “GoReplay issue: how do you deal with user session to replay the traffic correctly?” 2015. [Online]. Available: https://github.com/buger/goreplay/issues/154

[81] G. Amvrosiadis and V. Tarasov, “Filebench: a filesystem and storage benchmark.” [Online]. Available: https://github.com/filebench/filebench

[82] N. Brown, “Control groups series,” 2014. [Online]. Available: https://lwn.net/Articles/604609/

[83] L. Bugaev, “GoReplay: an open-source tool for capturing and replaying live HTTP traffic into a test environment in order to continuously test your system with real data.” [Online]. Available: https://github.com/buger/goreplay

[84] L. Capitulino, “Automatic memory ballooning,” 2013. [Online]. Available: http://www.linux-kvm.org/images/5/58/Kvm-forum-2013-automatic-ballooning.pdf

[85] J. J. Casey Schaufler, “Namespacing & stacking the LSM,” 2017. [Online]. Available: http://www.linuxplumbersconf.org/2017/ocw/sessions/4768.html

[86] J. Corbet, “Securely renting out your CPU with Linux,” 2005. [Online]. Available: https://lwn.net/Articles/120647/

[87] ——, “Network namespaces,” 2007. [Online]. Available: https://lwn.net/Articles/219794/

[88] ——, “Sysfs and namespaces,” 2008. [Online]. Available: https://lwn.net/Articles/295587/

[89] ——, “Seccomp and sandboxing,” 2009. [Online]. Available: https://lwn.net/Articles/332974/

[90] ——, “Cleancache and frontswap,” 2010. [Online]. Available: https://lwn.net/Articles/386090/

[91] ——, “Integrating memory control groups,” 2011. [Online]. Available: https://lwn.net/Articles/443241/

[92] ——, “Better active/inactive list balancing,” 2012. [Online]. Available: https://lwn.net/Articles/495543/

[93] ——, “vmpressure_fd(),” 2012. [Online]. Available: https://lwn.net/Articles/524742/

[94] ——, “Yet another new approach to seccomp,” 2012. [Online]. Available: https://lwn.net/Articles/475043/

[95] ——, “Extending extended BPF,” 2014. [Online]. Available: https://lwn.net/Articles/603983/

[96] ——, “The trouble with dropping groups,” 2014. [Online]. Available: https://lwn.net/Articles/621612/

[97] ——, “User namespaces and setgroups(),” 2014. [Online]. Available: https://lwn.net/Articles/626665/

[98] ——, “Filesystem mounts in user namespaces,” 2015. [Online]. Available: https://lwn.net/Articles/652468/

[99] ——, “Tracking actual memory utilization,” 2015. [Online]. Available: https://lwn.net/Articles/642202/

[100] ——, “Controlling access to user namespaces,” 2016. [Online]. Available: https://lwn.net/Articles/673597/

[101] ——, “Memory control group fairness,” 2016. [Online]. Available: https://lwn.net/Articles/684926/

[102] ——, “Network filtering for control groups,” 2016. [Online]. Available: https://lwn.net/Articles/698073/

[103] ——, “Writing your own security module,” 2016. [Online]. Available: https://lwn.net/Articles/674949/

[104] ——, “Containers as kernel objects,” 2017. [Online]. Available: https://lwn.net/Articles/723561/

[105] ——, “Namespaced file capabilities,” 2017. [Online]. Available: https://lwn.net/Articles/726816/

[106] V. Davydov, “Idlememstat: a simple utility for estimating idle memory size.” [Online]. Available: https://github.com/locker/idlememstat

[107] ——, “idle memory tracking,” 2015. [Online]. Available: https://lwn.net/Articles/643578/

[108] S. Derr, “Cpusets.” [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt

[109] J. Edge, “Freezing filesystems and containers.” [Online]. Available: https://lwn.net/Articles/287435/

[110] ——, “An introduction to SELinux,” 2004. [Online]. Available: https://lwn.net/Articles/103230/

[111] ——, “LSM stacking (again),” 2010. [Online]. Available: https://lwn.net/Articles/393008/

[112] ——, “Device namespaces,” 2013. [Online]. Available: https://lwn.net/Articles/564854/

[113] ——, “Control group namespaces,” 2014. [Online]. Available: https://lwn.net/Articles/621006/

[114] ——, “Namespaces in operation, part 7: Network namespaces,” 2014. [Online]. Available: https://lwn.net/Articles/580893/

[115] ——, “Progress in security module stacking,” 2015. [Online]. Available: https://lwn.net/Articles/635771/

[116] ——, “A seccomp overview,” 2015. [Online]. Available: https://lwn.net/Articles/656307/

[117] ——, “Filesystem images and unprivileged containers,” 2016. [Online]. Available: https://lwn.net/Articles/700422/

[118] ——, “On the way to safe containers,” 2016. [Online]. Available: https://lwn.net/Articles/700697/

[119] ——, “Container-aware filesystems,” 2017. [Online]. Available: https://lwn.net/Articles/718639/

[120] J. Fong, “Are containers replacing virtual machines?” 2018. [Online]. Available: https://blog.docker.com/2018/08/containers-replacing-virtual-machines/

[121] J. Frazelle, “Two objects not Namespaced by the Linux Kernel,” 2017. [Online]. Available: https://blog.jessfraz.com/post/two-objects-not-namespaced-linux-kernel/

[122] B. Gregg, “perf examples.” [Online]. Available: http://www.brendangregg.com/perf.html

[123] ——, “Working set size estimation.” [Online]. Available: http://www.brendangregg.com/wss.html

[124] S. E. Hallyn, “uts namespaces: Introduction,” 2006. [Online]. Available: https://lwn.net/Articles/179345/

[125] T. Heo, “cgroups v2.” [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v2.txt

[126] ——, “State of cpu controller in cgroup v2,” 2016. [Online]. Available: https://lkml.org/lkml/2016/8/5/368

[127] E. Horschman, “Hypervisor memory management done right,” 2011. [Online]. Available: https://blogs.vmware.com/virtualreality/2011/02/hypervisor-memory-management-done-right.html

[128] J. Johansen, “AppArmor by john johansen,” 2016. [Online]. Available: https://youtu.be/xahVJB2FxK0

[129] M. Kerrisk, “CAP_SYS_ADMIN: the new root,” 2012. [Online]. Available: https://lwn.net/Articles/486306/

[130] ——, “Hierarchical reclaim for memory cgroups,” 2012. [Online]. Available: https://lwn.net/Articles/516535/

[131] ——, “LinuxCon Europe: The failure of operating systems and how we can fix it,” 2012. [Online]. Available: https://lwn.net/Articles/524952/

[132] ——, “Stepping closer to practical containers: "syslog" namespaces,” 2012. [Online]. Available: https://lwn.net/Articles/527342/

[133] ——, “User namespaces progress,” 2012. [Online]. Available: https://lwn.net/Articles/528078/

[134] ——, “Anatomy of a user namespaces vulnerability,” 2013. [Online]. Available: https://lwn.net/Articles/543273/

[135] ——, “Namespaces in operation,” 2013. [Online]. Available: https://lwn.net/Articles/531114/

[136] ——, “Namespaces in operation, part 2: the namespaces api,” 2013. [Online]. Available: https://lwn.net/Articles/531381/

[137] ——, “Namespaces in operation, part 3: Pid namespaces,” 2013. [Online]. Available: https://lwn.net/Articles/531419/

[138] ——, “Namespaces in operation, part 4: more on pid namespaces,” 2013. [Online]. Available: https://lwn.net/Articles/532748/

[139] ——, “Namespaces in operation, part 5: User namespaces,” 2013. [Online]. Available: https://lwn.net/Articles/532593/

[140] ——, “Namespaces in operation, part 6: more on user namespaces,” 2013. [Online]. Available: https://lwn.net/Articles/540087/

[141] ——, “Mount namespaces and shared subtrees,” 2016. [Online]. Available: https://lwn.net/Articles/689856/

[142] ——, “Mount namespaces, mount propagation, and unbindable mounts,” 2016. [Online]. Available: https://lwn.net/Articles/690679/

[143] G. Klok, “Modern infrastructure,” 2017. [Online]. Available: https://www.youtube.com/watch?v=IVRlPkqYNBY

[144] A. Kopytov, “Sysbench: scriptable database and system performance benchmark.” [Online]. Available: https://github.com/akopytov/sysbench

[145] K. Korotaev, “Ipc namespace,” 2006. [Online]. Available: https://lwn.net/Articles/187274/

[146] O. Laadan, “Devicenamespace.” [Online]. Available: https://github.com/Cellrox/devns-patches/wiki/DeviceNamespace

[147] P. Menage, “cgroups.” [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[148] J. Morris, “Namespacing in SELinux,” 2018. [Online]. Available: https://blog.namei.org/2018/01/22/lca-2018-kernel-miniconf-selinux-namespacing-slides/

[149] P. Pandit, “rdma controller support,” 2016. [Online]. Available: https://lwn.net/Articles/674161/

[150] R. Rosen, “Understanding the new control groups api,” 2016. [Online]. Available: https://lwn.net/Articles/679786/

[151] R. van Riel, “vmscan: split lru lists into anon and file sets,” 2008. [Online]. Available: https://lkml.org/lkml/2008/6/11/276

[152] A. Vorontsov, “vmpressure: Linux VM pressure notifications,” 2012. [Online]. Available: https://lwn.net/Articles/524299/

[153] J. Weiner, “memcg naturalization,” 2011. [Online]. Available: https://lwn.net/Articles/442615/

[154] ——, “refault distance-based file cache sizing,” 2012. [Online]. Available: https://lwn.net/Articles/495423/

[155] ——, “memdelay: memory health metric for systems and workloads,” 2017. [Online]. Available: https://lkml.org/lkml/2017/7/27/429

[156] ——, “psi: pressure stall information for CPU, memory, and IO v2,” 2018. [Online]. Available: https://lwn.net/Articles/759658/

[157] R. J. Wysocki, “Freezing of tasks.” [Online]. Available: https://www.kernel.org/doc/Documentation/power/freezing-of-tasks.txt

INDEX

active list, 41
anon page, 43

batched jobs scheduling, 15
bit accessed, 36, 42
bit active, 42
bit referenced, 42

cache, 36, 39
capabilities, 18
cgroup, 37
checkpointing,migration, 15
classid, 17
cloud,cloud computing, 7
container, 13

dataset, 36
device files, 16
DRAM, 35

event counter, 38

fadvise, 14, 52
FIFO, 41
file page, 43
firewall, 17
full virtualization, 13

GUID, 18

hard limit, 38, 40
hypervisor, 11

idle page tracking, 42
idle ratio, idle page tracking, IR, 67
inactive list, 41
ionice, 14

IPC, 20

LRU, 41
lru, 41

MAC,mandatory access control, 16
madvise, 14, 52
Magency, 8, 9, 54
major page fault, 38
max limit, 38, 40
memory pool, 41
min limit, 38, 40
mlock, 14
MMU, 36, 42

nice, 14
numactl, 14

on demand, 31, 54
OOM, 38, 40, 44
overselling,overbooking, 7

page counter, 39
page fault, 38
page in, 38
page out, 38
page table, 36
paravirtualization, 13
perf, 17
PFRA, 38, 41
pglost, 58, 90
pgstolen, 58, 90
PID, 19
pid, 16
prioidx, 17
privilege escalation, 18

refault distance, 48
refrigerator, 16
rotate ratio, 46

scan priority, 45, 49
soft limit, 41, 52
stat counter, 39
swapiness, 45

taskset, 14
traffic control, 17

UID, 18

virtual address, 36
virtual memory, 36
VM, Virtual Machine, 12
vmpressure, 48

workingset, 36