INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2023IPPAS021

Thèse de doctorat

Improving memory usage in virtual machines

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 14/12/2023, par

YOHAN PIPEREAU

Composition du Jury :

Pierre Sens Professeur, Sorbonne Université (LIP6)	Rapporteur
Romain Rouvoy Professeur, Université de Lille (CRIStAL)	Rapporteur
Brice Goglin Directeur de Recherche, Centre INRIA de l'Université de Bordeaux	Président du jury
Stella Bitchebe Chercheuse postdoctorale, McGill University (DISCS lab)	Examinatrice
Gaël Thomas Professeur, Institut Polytechnique de Paris	Directeur de thèse
Mathieu Bacou Maître de Conférence, Institut Polytechnique de Paris	Co-encadrant de thèse
Jean-Pierre Lozi Chargé de Recherche, INRIA Paris	Invité

Improving memory usage in virtual machines

v3.0 (final version)

Yohan Pipereau

January 1, 2024

À Koko et Julie

Remerciements

J'aimerais tout d'abord remercier mon directeur de thèse, *Gaël Thomas* pour la proposition originale d'un sujet d'une importance majeure pour les infrastructures Cloud ainsi que pour la relecture du manuscrit.

Je voudrais ensuite remercier *Mathieu Bacou* pour avoir rejoint l'encadrement de thèse à mi-chemin ainsi que pour sa participation à l'écriture des papiers et la conduite des expériences sur les plateformes FaaS.

Je remercie également *Jean-Pierre Lozi* qui nous a rejoint un peu plus tard pour son aide à la préparation de deux soumissions de papiers et pour nous rappeler à l'essentiel et nous orienter grâce à son expertise.

Je suis également reconnaissant à *François Trahay* pour son aide à la préparation de notre seconde soumission en conférence ainsi que pour l'évaluation du placement de page de VMs sur les infrastructures NUMA. Je ne peux conclure mes remerciements à François sans le remercier pour sa méthodologie de construction du plan de l'état de l'art, qui m'a valu quelques critiques, mais qui m'a permis de livrer rapidement un ensemble cohérent.

Je souhaiterais ensuite remercier *Pierre Sens* et *Romain Rouvoy* pour leurs lectures attentives de ce manuscrit ainsi que pour leurs retours précieux pour améliorer l'organisation et le contenu du manuscrit. Je remercie également tous les membres de mon jury d'avoir accepté de participer à celui-ci : *Stella Bitchebe* et *Brice Goglin*.

Je ne saurais assez remercier *Denis Conan* pour son aide et ses conseils dans le traitement statistiques de données expérimentales afin de mettre en avant la significativité des effets NUMA, mais aussi pour son partage sans condition d'outils et de méthodes scientifique. Je suis reconnaissant de m'avoir incité à construire des abstractions qui m'ont permis de comprendre différents problèmes et de m'avoir rappelé à la démarche scientifique lorsque je m'en éloignais. De manière plus importante encore, je souhaiterais le remercier pour avoir régulièrement pris de mes nouvelles au cours de l'écriture du manuscrit et pour avoir trouvé les mots réconfortants lors du retour de nos soumissions.

Merci également à *Pierre Sutra* pour ses retours, et pour nos discussions sur mes travaux, et en particulier pour m'avoir aidé à corriger l'introduction et la conclusion de ce manuscrit.

Un grand merci à *Amel Bouzeghoub* et *Sophie Chabridon* pour avoir servi de

boussoles en fin de thèse, en discutant aussi bien du calendrier de fin thèse, de la préparation du manuscrit que des poursuites possibles après la thèse.

Je souhaiterais remercier de nombreux membres du département RST, de l'équipe PDS, du département INF ainsi que de la direction de la recherche pour leur soutien moral, méthodologique, le partage d'information et pour les bons moments passés à vos côtés au cours des dernières années. La liste des personnes à remercier est très longue mais je souhaiterais en particulier remercier *Elizabeth Brunet*, *Olivier Levillain*, *Sophie Chabridon*, *Chantal Taconnet*, *Bruno Defude* et *Hervé Debar* ...

De plus, je souhaiterais remercier plusieurs générations de doctorants pour l'aide méthodologique, scientifique et leur soutien moral pendant ces années. En particulier, je souhaiterais remercier *Alexis Lescouet* pour ces conseils précieux et l'aide qu'il m'a apporté dans la compréhension de divers mécanismes des systèmes d'exploitation. À *Adam Chader* pour son énergie débordante, son optimisme inépuisable, et sa joie de vivre constante qui a égayé de nombreuses journées lors de l'écriture du manuscrit. C'est un plaisir de pouvoir discuter ensemble de problèmes, design, algorithmes ou mécanismes sans jugements de valeur.

À *Marie Reinbigler* pour m'avoir inspiré à discuter avec des personnes en dehors de l'équipe, pour avoir partager des outils et des méthodes d'analyse de résultats, pour son soutien moral. À *Mickaël Boichoit* pour son soutien, pour sa bienveillance et pour nos multiples discussions autour de nos problèmes communs d'interconnects, et parce qu'être un peu fou de temps en temps c'est chouette aussi. À *Subashiny Tanigassalame* pour son soutien depuis le premier jour et pour sa confiance à collaborer ensemble sur un sujet passionnant. À *Anatole Lefort* pour ses conseils dans l'écriture de cette thèse, pour la confiance qu'il m'a accordée pour travailler ensemble et pour les bons moments que nous avons passés ensemble en dépit des divers confinements et restrictions. Je voudrais aussi remercier *Alexis Colin*, *Boubacar Kane*, *Damien Thenot*, *Pedro Borges* et *Rémi Dulong* pour les bons moments passés ensemble, malheureusement, trop limités par le télétravail et le COVID.

Plus généralement, je souhaiterais remercier mes anciens professeurs, mes anciens directeurs de stages, mes amis, qui m'ont aidé et soutenu durant cette thèse. En particulier, je voudrais remercier *Alexandre Conte*, *Thibaut Sautereau*, *Vincent Gouvernec*, *Vincent Brillault*, *Quentin Bouget*, *Jérome Tollet*, *Clément Parssegny*, *Éloïse Bonnet*, *Hamza Benfkira*, *Jules Gonzales*, *Anna-Rose Lescure*, *Florian Grante*, *Victor Védie*, *Romain Cherré*, *Stevan Coroller* et tous ceux que je n'ai pas pu citer faute de place.

Il aurait été compliqué de préparer ce doctorat sans l'aide de ma famille. En particulier, je dois beaucoup à mes *parents*, à mes *grands-parents*, à mon *frère* ainsi qu'au *reste de ma famille* pour leur encouragement et leur confiance. Je voulais vous remercier de m'avoir poussé à être curieux et persévérant ainsi que pour m'avoir soutenu tout au long de cette thèse même dans les moments les plus difficiles.

Ces remerciements ne peuvent s'achever sans remercier *Julie Chevrier* pour son soutien moral quotidien, mais aussi pour avoir relu mon manuscrit et prêter une oreille attentive aux différents problèmes rencontrés. Je suis heureux d'avoir pu partager des moments parfaits à tes côtés en parallèle de la réalisation de mes travaux. Après quatre longues années, je suis reconnaissant pour tous tes efforts afin de faire tenir notre relation en dépit d'heures de travail parfois compliquées et

des ajustements d'emploi du temps à la veille de soumissions ou d'autres échéances.

Contents

Introduction

Context

Datacenters host a large number of servers, and they offer software services or directly rent access to a subset of their physical resources to customers. In order to rent independent subset of physical resources, datacenters rely on different techniques which support program execution while ensuring isolation of resources between each others. Resource isolation is mostly enforced with the help of virtual machines, which are used to enforce confidentiality, integrity and provide mechanisms to prevent resource stealing. In addition to offering resource isolation, virtual machines are also convenient deployment units. Indeed, on the one hand, virtual machines support the execution of *any* operating system and software stack. On the other hand, virtual machines supports migration from one server to another which is hard to support in other execution units because of dependency conflicts between source software stacks and destination software stacks during migration. Virtual machines tackle this problem by shipping end-to-end software stacks (applications, libraries, operating system) to prevent dependency conflicts.

Selling shared access to a set of physical resources through virtualization has made datacenters highly profitable over the past years. However, multiple cloud providers have reported that resources in their datacenters remain largely underused. This is caused by the virtual machine abstraction which requires static allocation of server resources at start time. Static provisioning poses major constraints.

First, programs usually have dynamic resource usage over time and resource usage changes a lot especially in applications exposed to external inputs (web-servers, load-balancers, ...). While resource usage is dynamic, on the contrary, resources are allocated statically when the virtual machine is started. This means that allocated resources which are unused by the VM can not be used for any other execution. Cloud providers observe these resources as a potential source of additional profits even though using them while maintaining safety and performances is a challenge.

Second, since resources are mostly static, the declaration of virtual machine resources before provisioning is complex as it must forecast worst case scenario of resource usage. Most declarations end up overprovisioning their instances to ensure execution safety and prevent crashes. Overprovisioning VMs amplifies even more the amount of unused resources in the datacenter.

Third, since virtual machines can only be executed on a single server at a time

and may have different allocation requirements, finding an available allocation server candidate is challenging. Indeed, cloud providers try to find an allocation configuration of VMs on multiple servers which reduces the remaining amount of unallocated resources on each server. This configuration known as the optimal configuration is a well-known problem of combinatorial optimization which directly maps to the knapsack problem known to be NP-complete.

In this thesis, we focus exclusively on improving usage of memory resources, and we leave improvement of storage and processing power as future contributions. The first motivation to focus on memory is that memory represents a significant part of datacenter costs. Indeed, in 2023, Microsoft Azure has reported that memory can account for up to 50% of a datacenter costs [50]. The second reason to focus on optimizing memory usage in VMs is that new hardware is being released which could help reduce datacenter cost. Indeed, since 2015, there has been multiple proposals to build a new rack infrastructure with a high-speed interconnect to support remote memory accesses. Supporting remote memory accesses is expected to be a game changer for virtual machines as it enables to access memory fragmentation leftovers and potential unused pool of memory in remote servers.

We propose to target specific directions to reduce memory usage in datacenters running virtual machines

In particular, we try to identify the challenges introduced by virtualization to use disaggregated memory. We identify that it is important to require as few changes as possible in applications executing inside VMs to maintain transparency. Furthermore, while the use of remote memory should help VM scheduling decisions, it is important to maintain a low impact on application performances.

A first research direction will propose a solution to use virtualization with disaggregated memory and to study the impact on application performances.

A second research direction will try to identify the motivations behind static provisioning of VMs. It will investigate the reasons behind the low adoption of VMs adapting their memory capacity during their execution.

A third research direction will focus on identifying new design solutions to implement efficient mechanisms for dynamic memory changes in VMs. This direction will investigate the performance impact of time-sharing memory resources between VMs and the consequences in memory usage.

Contributions

The contributions of this thesis are as follows:

- We implement and evaluate **ODswap**, a transparent mechanism to support transparent remote memory accesses in VMs based on Linux swapper mechanism and RDMA, a low-latency fabric. ODswap addresses the problem of overconsumption of memory on the remote server and uncollaborative page placement caused by virtualization semantic gap. These problems are addressed in ODswap by implementing on-demand allocation on remote memory and paravirtualization to use guest memory management information. We evaluate ODswap to determine the impact on applications performances, to

compare this architecture with distributed shared memory and cover how far memory can avoid virtual machine crashes in case of peak memory usage.

- We present the results of several experiments on existing VM mechanisms related to **memory heterogeneity** and **dynamic capacity changes**. In particular, we show how the design of existing VMs and hypervisor is inefficient to provide fast memory capacity changes. We also demonstrate and clarify the semantic gap between guest memory management and hypervisor memory management which leads to suboptimal page placement on tiered memory.

- We present **ExoVM**, a work-in-progress prototype to support fast dynamic changes of available memory in VMs. ExoVM is strongly motivated by a large evaluation of existing mechanisms to support dynamic memory changes which exhibit extremely low speed in existing solutions. ExoVM proposes to directly let a virtual machine initiate a memory reconfiguration request to the hypervisor and proposes solution to let the hypervisor control memory usage. Early evaluation results show that ExoVM is a promising approach to limit memory fragmentation and to improve usage of memory over time in a single server.

Organization of the document

The thesis is organized as follows: part I presents historical and recent advances in management of heterogeneous memory in operating systems as well as a large overview of virtual machines from low-level mechanisms to their consequences and constraints for resource usage in the datacenter. In particular, chapter 1 presents the emergence and characterization of various heterogeneous memory backends. Chapter 2 presents the internals and foundations of memory management in Linux operating system. Chapter 3 introduces new proposals for memory management on various emerging memory backends. Chapter 4 focuses on the use of far memory in different systems such as databases, language runtimes and operating systems. Chapter 5 reviews how modern system virtual machines are implemented and the consequences for datacenter orchestration.

Next, the following chapters presents the main contributions of this thesis. Chapter 6 presents our first prototype, ODswap. Chapter 7 discusses multiple detailed evaluations of existing solutions to adapt memory capacity in VMs to motivate the design of ExoVM, the second contribution of this thesis. Chapter 8 presents our second ongoing prototype ExoVM. Finally, we conclude this thesis with the main takeaways and future works.

Part I

Support for memory disaggregation in virtual environment

Over the last few years, cloud providers have been trying to execute as much customer jobs as possible on a reduced number of servers to make higher profits. They commonly run these customer jobs in execution units such as containers and virtual machines to isolate deployments and manage a uniform view of customer jobs through abstracted services. However, improving datacenter resource usage remains a hard problem with a wide range of solutions proposed from hardware to orchestration software stacks, which still leaves significant percentage of resources unused.

The decreasing cost of memory capacities enable to increase server capacities at similar cost while executing more jobs. However, this solution is unsatisfactory since resources remain largely underused. A more promising approach is brought by new hardware enabling rack-scale memory access with low-latency and high-throughput. This approach is expected to be a game changer in the cloud landscape. Indeed, accessing remote memory resources would enable resource upgrades at a lower cost by avoiding entire server changes. Accessing larger pools of memory would also ease virtual machine placement algorithms. However, remote memory accesses may hurt end-user performances compared to local memory accesses, and requires software stacks to propose clever placement of memory.

Leveraging new memory technologies in virtual machines require changes to various software layers. Indeed, the virtual machine abstraction is tightly dependent on existing operating systems services such as memory and IO management and scheduling. Yet, they also require new OS services to leverage hardware accelerators in processor, which help to speedup IO and memory management. Finally, a user-space application named hypervisor is also required to complete the set of mechanisms proposed by operating systems and to help run the VM abstractions in clusters of hypervisors.

This part first proposes a review of existing and new memory backends and their specificities. Then, it dwells on newer approach to memory management and proposes a review of Linux kernel basic memory management services and abstractions. After getting a better understanding of OS level memory management, we review the existing solutions for handling memory heterogeneity at operating system level with a focus on legacy heterogeneous backend such as storage backends and NUMA memory. Next, we review recent work on far memory and disaggregated memory in different software systems such as databases, language runtime and operating systems with a focus on the different hardware assumptions and requirements proposed. Finally, we present legacy and recent advances in virtual machine resource management by reviewing challenges and solutions at host, rack and datacenter levels with a focus on memory resources at hypervisor level.

1 Heterogeneous memory backends

The landscape of memory hardware has changed a lot over the last twenty years with a large spectrum of trade-off between each backend. Operating systems and software stacks are relying heavily on memory management abstractions inherited from the Unix operating systems. However, the large gains in throughput with High-Bandwidth Memory (HBM), the arrival of byte-addressable persistent memory (NVDIMM), the low latency accesses to flash drives (NVMe) is driving system and database communities to reconsider existing abstractions. In this section, we propose a quick characterization of new memory backends from a software perspective. First, we present trends in memory hardware and its usage in computers. Second, we discuss memory heterogeneity and the different properties of memory backends. Then, we review some common memory backends from a software perspective before discussing the tight coupling of interconnects and cache coherency for transparent remote memory communications.

1.1 Trends in memory usage

Compute requires storing intermediate results somewhere. Processors rely on a limited set of registers to store intermediate results. Because of the limited size offered by these registers, program mostly use these registers to store temporary results and rely on larger memory technologies to store more information.

Additionally, some program results need to survive the program lifetime and thus require the need of persistent storage. Historically, computers have mostly relied on two types of memory technology: volatile byte-addressable memory and persistent block storage. Storage has been referring to the class of memory technology offering persistence guarantees and working at block granularity. However, the arrival of byte-addressable non-volatile memory is redefining high-level understanding of these hardwares. This section presents the two main memory trends which are the convergence of storage and memory in terms of performance and properties and memory speed becoming a bottleneck over CPU speed in many programs.

1.1.1 Filling the latency gap between storage and memory

On the memory side, volatile memory as it exists in modern computers appeared in late 1960s by using semiconductors to store information. Over time, memory capacity have increased considerably from a few bytes to hundreds or thousands of gigabytes of memory in more recent servers.

On the storage side, before the arrival of NAND flash drives, persistent storage has relied for a long time on hard drives and tape drives. These storage solutions require mechanical mechanisms to seek data incurring long seek latencies. In order to fill the gap between storage and memory latencies, storage vendors have started NAND flash based on semiconductors similarly to volatile memory. Thus, recent storage solutions using NVMe protocol for communication between storage controller and the target can even yield sub 100 µs latencies blurring the latency difference between volatile memory and storage.

1.1.2 Memory wall and the increasing latency difference between compute and memory

In 1995, Wulf et al.[165] first reported a divergence between exponential growth of processor speeds and memory latency and bandwidth improving at a slower pace. Under cache hit, memory access goes approximately to processor speeds, while cache miss exhibits memory access latency. Even considering good cache hit ratio (from 99 % to 99.8 %), the number of CPU cycles grows from a few cycles in the 1990s to around a hundred in the 2010s. Their conclusion is known as the *memory wall* and forecast that memory latency will bottleneck all application performances in the next years.

The memory wall constraint has motivated application rewriting to improve cache accesses. However the number of CPU cycles upon cache miss has remained high.

Similarly to volatile memory accesses other storage backends have become considerably slower compared to the improvement in CPU speeds. An entire spectrum of access latencies expressed in CPU clock cycles is now available with higher capacities available as latencies become higher.

1.2 Heterogeneous properties of memory backends

In section 1.1, we have described some of the general trends in memory and storage evolution. In order, to fully encompass the difference between each backend, we propose to review core properties variations between each backend. Indeed, even though memory heterogeneity is commonly seen as latency (§1.2.1) and throughput (§1.2.2) difference, there exists multiple other properties to characterize each backend such as parallelism, granularity, capacity, cost and power consumption.

1.2.1 Read/Write Latency

There are significant order of magnitude between the different memory latencies [1]. For example, commonly reported order of magnitudes for L1 SRAM access is around 1ns while main memory access is around 60 ns, RDMA accesses are around 1 µs and SATA SSD 1 ms. These latencies variations appear for various reasons. One reason is that memory backend technologies (SRAM, DRAM, 3DXpoint, ...) introduce significant latency gaps. Another reason is that memory devices are used at varying distance to where computation occurs. Furthermore, some backends exhibit differences between read and write operations. For example, NVDIMMs write latency is similar to DRAM write latency while read latency is clearly slower than DRAM read latency.

1.2.2 Read/Write Bandwidth

Interestingly latency is not the only important performance metric for application efficiency. Some applications may be latency-bound while others are bandwidth-bound. Typically, HPC applications leveraging GPGPU with matrix computations often need to load a large volume of data (matrix) at a time. High bandwidth memory (HBM) fits the need for high bandwidth need by offering between a few hundreds of gigabytes per second to a terabyte per second of bandwidth compared to classic DRAM, which bottlenecks around a hundred gigabyte per second for 6 channels DDR4.

1.2.3 Parallelism

IO parallelism has become a core property of attached storage and memory solution. Memory has provided IO parallelism for a long time. However, IO have a long history of sequential accesses caused by hard drives disks reading head. Legacy IO controllers stacks (SCSI, SATA) have been built on these sequential access assumption. Recently, storage has adopted flash memory storage and parallel IO controllers (NVMe) exposing multiple hardware queue. While IO parallelism offers large performance gain it also causes various challenges for the software developer. Indeed, since IO requests and IO completions may be reordered, it violates the assumption that IOs are delivered in program order as guaranteed by serial backends before.

1.2.4 Granularity

Access granularity may be *fixed-size* or *dynamic-size*. Fixed size management enables easier tracking and management while dynamic sizes permits better packing and resource usage. Access granularity may be performed at word size, cache line size, sector size, page size. Appropriate granularity is a key criterion when performing accesses. Using too-small granularity leads to multiple round-trips causing higher IO latency, while too-large granularity leads to unnecessary longer IO transfers and resource waste referred as *IO amplification* [125].

[1] Here latency is expressed as latency from CPU to DIMM which better reflects application latency rather than the latency from the DRAM controller to memory cells (CAS latency)

1.2.5 Capacity

Capacity is an immediate property of a storage solution. A general rule of thumb is that larger capacity comes with higher latencies. Capacity of a memory backend is limited by price considerations thus capacity is commonly considered as a $GiB/\$$.

1.2.6 Cost

Cost is also a key factor of adoption for a memory technology. Computers usually balance costs by adapting parameters such as *throughput*, *latency*, *capacity*. Changing these parameters can sometimes be achieved directly by computer upgrades. However, some parameters (latency, throughput) can only be changed indirectly by selecting different vendors for an identical technology, by changing the memory technology directly (SRAM vs DRAM) , by picking different generations of a backend (e.g. DDR4, DDR5).

1.2.7 Power

Power consumption is becoming a key property in datacenter and supercomputers because of various limits to the use of energy. In particular, datacenters must take into account various factors related to energy supply such as energy cost, maximum power available. Additionally, there exists huge variations across countries regarding energy supply (e.g. carbon footprint, variable energy supply) which directly impact datacenters. New computers must become as *energy efficient* as possible with worldwide energy cost increase and supercomputers bottlenecked by power supply in their quest for zettascale. Supercomputers in TOP 500 are approaching zettaflop (10^{21} floating point operations per second) and their energy consumption is far above a single nuclear plant reactor (around 1000MW) [106]. Dan Ernst [50] justifies the industrial success of a memory technology based on power consumption (W) and cost ($) at identical performance goals. He explains the adoption of HBM in the HPC landscape because HBM2 and HBM3 generations enables 6 to 8 MW for 200 PB/s while DDR4 and DDR5 require 47 to 55 MW to achieve the same bandwidth. More and more backends and interconnect systems are built with power and heat as constraints and rely on low power mode [111, 82] to reduce both.

1.2.8 Additional properties

Many other memory properties such as *pattern performances* (random vs sequential), *endurance* (SSD wear-leveling), *Error correction* (ECC), *data replication* (RAID devices), *data persistence*, bandwidth evolution under *concurrency* (scalability with the number of threads) or *encryption* (SGX) exist though these are not detailed because they are less relevant for our study of Heterogeneous memory.

In this section, we have reviewed some of the properties of memory and IO backends. We have illustrated these properties with different memory backends. Memory heterogeneity is often observed as differences in the average latency of all operations on the backend. These limited view of heterogeneity is reflected in memory topology

tables (see §3.1.1) which only report a subset of the various properties of heterogeneous memory. However, heterogeneity can be captured notably in the differences between load and store latencies, in load and store throughput, in the backend parallelism, its lifetime, persistence guarantees, and operation granularity. Additionally to these properties, which directly impact the developer, power consumption and cost have also been shown to impact the success of a memory backend and guide its use in the datacenter. All these properties are the result of hardware implementation choices in communication facilities and memory backends which are presented in the next two sections.

1.3 A review of common memory backends

In this section, we present some of the memory backends available in modern computers to review how their properties is best leveraged. There exists other backends, which are still being developed (Phase-change memory, Z-RAM) which are not reviewed in this section because many things are left unclear regarding their future adoption. Moreover, there exists memory backends that target specific use cases (LPDDR for mobile, GDDR for GPU), which are outside the scope of this thesis. We rather focus on CPU caches memory, byte-addressable volatile memory, high-bandwidth memory, byte-addressable persistent memory backends in a first time. Then, we review memory backends based on interconnect networks or fabric networks with non-uniform memory architectures (NUMA) and remote direct memory accesses (RDMA) since the use of these networks introduce new heterogeneity challenges for software developers.

1.3.1 Byte-addressable volatile memory

Random access memory is the most common volatile memory backend used in servers. CPU issues accesses to RAM using addresses. These addresses are translated to a physical storage unit named *memory cells* using a component named address decoder. Thus, RAM can be represented as a matrix of memory cells accessible using row and column identifiers. The remainder of this section presents multiple Random Access Memory available suited for different use cases such as SRAM for CPU caches and SDRAM for main memory.

1.3.1.1 SRAM, CPU caches memory

Static RAM (SRAM) is a low latency and low power RAM technology. Data stored in a SRAM memory cell remains available with no refresh required. The main drawback of SRAM is that memory cells are made of 4 to 6 transistors which impose high cost for lower storage capacity.

SRAM is commonly used today for CPU caches because of its high cost-per-capacity but high-bandwidth and low latency.

1.3.1.2 DRAM, early main volatile memory

Contrarily to SRAM, Dynamic Random Memory Accesses (DRAM) relies on different electronic components named capacitors instead of transistors. Capacitors enable to reduce the cost of memory capacity however charges in capacitors leaks over time and thus require refreshing contrarily to SRAM. DRAM refreshes consume more power and causes slower memory accesses than SRAM. This refreshes are scheduled by the *DRAM controller* which tries to avoid conflict with ongoing reads and writes.

In order to provide higher storage capacity, DRAM also needs to reduce the number of pins encoding an address. This is achieved using *Index multiplexing* by using two clock cycles to read a single address.

DRAM has been widely used in computers prior the 2000s as a backend for main volatile memory. However, main memory has become a bottleneck with processor speed increasing very quickly (see §1.1.2).

1.3.1.3 SDRAM and DDR RAM, main volatile memory

A problem with DRAM is that they were *asynchronous* meaning that the CPU required to access DRAM pins with its own time management. This caused the memory bus to be unsynchronized with DRAM bottlenecking high-data transfers for serial accesses. SDRAM has been proposed as a logical evolution of DRAM to increase memory speed and simplify the memory interface [94]. SDRAM proposes to share the bus clock between SDRAM, CPU and chipset at given frequency.

Moreover, SDRAM supports multiple banks on a single chip which enables to process multiple memory commands in parallel. Indeed, even if all banks share the same pins, each memory bank is able to treat memory commands in parallel which enables pipelining memory commands (e.g. precharge, row and column indices selection).

DDR RAM is an improvement over SDRAM which doubles the speed by transferring two data words per clock cycle instead of single word for SDRAM. It is being used as the de facto standard for main volatile memory in modern computers.

1.3.1.4 HBM, High-bandwidth memory

GPUs have become popular accelerator used for multiple high performance and AI workloads as a way to deliver efficient matrix computation. GPUs ability to deliver lots of operations per time unit also causes new challenges to store information with high bandwidth.

HBM leverages a new memory technology named *3D stacked memory* which stacks DRAM dies vertically to gain more memory density [99]. The high bandwidth of HBM memory is due to memory being stacked on an interposer component close to the processor (GPU) contrarily to GDDR (Graphics Double Data Rate) historically used in GPUs. The technique has now been used for DRAM too.

HBM has been commercially introduced for GPUs on AMD's Fiji GPUs and Nvidia's Pascal GPUs but also for CPU with Intel Xeon Phi Knights Landing (KNL) [128] famous for supporting AVX512 vectorized instruction set. It has been adopted in A64FX the ARM architecture for Fugaku supercomputer [129] with a reported bandwidth of 1024 GB/s. HBM has been adopted mostly for GPGPUs and CPU with Vector Extensions.

1.3.2 NVDIMM, byte-addressable persistent memory

NVDIMM is a new hardware released by Intel under the name of Intel Optane Memory leveraging 3D XPoint Memory. These DIMMs provide *byte-addressable persistence* with lower latency than NVMe but higher latency than DRAM. Intel Optane is more efficient (lower latency and higher bandwidth) for reads than for writes especially when thread concurrency increases [74]. This is due to efficient sequential prefetching with read-ahead. The access pattern is also important as

Izraelevitz et al.[74] reported that sequential access pattern can yield up to 2x better latency than random access pattern thanks to adjacent requests merging. Byte addressable persistence has led to many scientific and industrial contributions, but these are out of our scope. NVDIMMs have been discontinued in 2022 and are expected to be commercialized again with the advent of CXL.mem devices (Compute Express Link) in the next few years (see §1.5.3).

1.3.3 NUMA, byte addressable, cache coherent remote memory accesses

NUMA machines were first commercialized in the 1990s following the logic of scaling of the number of cores on processors. NUMA machines offer the possibility to aggregate in the same machine a set of nodes of CPU, memory and interconnect resources. It enables any NUMA node of the machine to access resources from another node at the cost of higher latency and reduced bandwidth. A set of NUMA machines known as ccNUMA quickly started to implement cache coherency protocol to allow processor from different nodes to access memory. Cache coherent NUMA machines are now the de facto commercially available NUMA machine.

1.3.4 SSD persistent storage

Solid State Drive (SSD) are a set of drives that only rely on electronic circuitry for persistent storage. They classically rely on Flash Memory, but Intel has also released Optane SSD leveraging 3D XPoint technology, which performs better at low access granularity [164]. SSD provide lower latencies than Hard Disk Drive which require long seek time for moving read head to particular locations on the disk. Initially, SSD relied on AHCI controllers to perform IO operations over a SATA bus. However, AHCI is a serial controller (single hardware queue), which performs sequential IO operations. Newer SSDs rely on NVMe interface on top of PCIe bus to offer parallel accesses to SSD.

1.3.5 RDMA, remote memory accesses using PCIe DMA

Remote Direct Memory Access (RDMA) refers to a set of protocols which can perform remote memory operations without involving the destination processor. Because of its high throughput and low CPU overhead, RDMA has been widely used in High Performance Computing frameworks such as Message Passing Interface (MPI). It is also widely used in storage systems for Distributed File Systems [101] or disaggregated storage such as iSCSI Extensions for RDMA (iSER) [73] and NVMeoF [112]. Most common RDMA protocols are *Infiniband*, *iWARP*, *RoCEv2*. Other implementations exist mostly for HPC use cases such as the *Tofu interconnect*. There exists different vendors of a protocol.

Networking protocols are built as layers composed using *encapsulation*. A classic model of networking distinguishes physical layer, link layer, networking layer and transport layer. Each networking layer offers different services.

One of the key feature of RDMA networking is that the entire networking stack from physical to transport layer is embedded into the hardware. This enables very efficient buffer delivery between hosts at the cost of flexibility.

Multiple evaluations [139, 88, 151] of RDMA latency and bandwidth are available for older cards generations. They show that RDMA cards are able to deliver close to 1 µs latency for one-sided operations of 64 B messages. Similarly to NVDIMM, RDMA may offer heterogeneous performances between read and writes operations. Kalia et al [80] and Herd [81] observe that RDMA only supports reliable reads but it can support both reliable and unreliable writes. Herd further observes that for messages under 256 bytes, unreliable and reliable writes offer 34 % higher throughput than reliable reads. Herd explains the difference because read operations need to maintain a state for the operation at RDMA level and PCIe levels in order to get the response. They observe a limited number of 16 concurrent read requests in their RNICs. They further notice that the average **unreliable RDMA write** operation latency is half the average latency of RDMA **reliable read operations** because it uses half the PCIe round trips since it does not require acknowledgement.

1.3.6 CXL

Compute Express Link (CXL) [34] is a recent industry standard uniting multiple chip vendors in a consortium giving it lot of credits. The consortium has proposed a specification for a cache coherent interconnects. CXL provides a unique standard to solve multiple challenges in the use of new hardware such as cache-based accelerators (SmartNICs) (**type 1 devices**), memory-based accelerators (ASIC, FPGA, GPGPU) (**type 2 devices**), and memory pools (**type 3 devices**)

There exist already three major versions of CXL. CXL 1.1 and CXL 2.0 rely on PCIe 5 while CXL 3.0 is expected to rely on PCIe 6.

1.3.7 Synthesis and order of magnitudes

Heterogeneous memory is defined by the variations in *IO Bandwidth* and *IO latencies* across the different backends. *Capacity* is an important aspect of an heterogeneous memory technology since it may condition the access semantics. *Parallelism* or *Access Granularity* also impact performances by supporting concurrent operations or by requiring smaller number of round-time-trips to transfer a message [80]. *Cost*, *Power* are important factor of adoption of heterogeneous memory technologies.

This memory heterogeneity is commonly represented as a triangle with the lowest access latency represented at the top of the triangle and highest access latency at the bottom of the triangle. Although this representation hides key performance metrics of memory heterogeneity such as read-write latency differences and the bandwidth advantages some memory offer with similar latencies. Thus, in Table 1.1, we propose another imperfect aggregation of data to represent memory heterogeneity.

Table 1.1 summarizes bandwidth and latency orders of magnitude aggregated from various sources for different memory technologies.

The figures provided in the table Table 1.1 only provide order of magnitude

	max read bandwidth (GiB/s)	max write bandwidth (GiB/s)	Avg random load latency (ns)	Avg random write latency (ns)
local DDR4 DIMM (6 channels)	120 [75]	140 [75]	80 [75]	90 [75]
remote DDR4 DIMM	40 [75]	15 [75]	180	180
Optane NVDIMM (6 channels)	7 [75]	2 [75]	300 [75]	90 [75]
Flash SSD (NVMe)	3.5 [164]	2.7 [164]	200,000 [164]	20,000 [164]
HBM2	600-1000 [124]	600-1000 [124]	100 [172]	100 [172]
RDMA Mellanox Connect-X3	0.3 [151, 81]	0.5 [151]	1000 [151, 81]	1000 [151, 81]
PCIe 3 DMA local mem	5-6 [108]	5-6 [108]	450 [108]	550 [108]
CXL.mem (with RAM)			300 [104]	300 [104]

Table 1.1: Read/Write latency for 64B random accesses of various memory tiers

however summarizing the differences in this table introduces a bias. Indeed, for a same memory backend different versions of hardware controllers, interconnects or fabrics introduce different results. Overall bandwidth and latency values may vary depending on the degree of operation concurrency [97].

Reported latency values are *average* while various work [97] made a point about application performances being highly correlated to tail latencies. Moreover, benchmarking independently read and write latencies hides bottlenecks introduced when these operations are performed concurrently because of serialization costs [97].

Extra care must also be granted to reported bandwidth values since bandwidth has scaled up very quickly for the past years by using more queues. Memory controllers (HBM, DRAM, ...) typically refer to queues as *channels* while PCIe refer to *lanes*. For instance, memory bandwidth for PCIe backends has doubled every few years and similarly DDR4 bandwidth grows linearly with the number of channels. Another influence is DDR frequency determining the number of commands it can perform per seconds.

Additional important operations exist for some of these backends like *erase* on storage backends or *atomic operations* in RDMA for which latency and bandwidth has not been reported. Finally, these figures have been aggregated from different papers using different hardware, software stacks (OS, libraries, ...).

1.4 Resource disaggregation

Now that we have presented the key trends in memory usage and main memory technology, this section reviews the interest of ***resource disaggregation****, a recent trend to improve resource usage in the rack. Resource disaggregation is the idea of isolating server resources into different resources connected through a high-speed and low-energy interconnect. Resource disaggregation is often considered as the separation of four class of resources: First, there are compute units which include CPU but also processing unit used as accelerators like GPGPU or TPUs. Second, there are memory resources which are becoming more diverse as described in section 1.3. Third, there are storage resources ranging with Magnetic Tapes, Hard Disk Drives and Solid State Drives with SATA or NVMe interfaces. Fourth, some scenarios have considered network disaggregation has a use case though it has been left mostly unexplored.*

In this section, we first review the benefits of using remote resources in datacenters to improve resource usage. Second, we discuss remote storage solutions that have been proposed since the 2000s and that have paved the way to memory disaggregation. Third, we review the proposal of a new rack-scale architecture around 2015 where compute, storage, memory and network resources are disaggregated in different servers. Fourth, we discuss a realistic architecture where servers maintain their set of resources (compute, memory, storage, networking) to be shared between servers in the rack.

1.4.1 The benefits of resource disaggregation

Resource disaggregation, whether it is storage disaggregation or memory disaggregation claims multiple benefits over current server architecture aggregating compute, storage, memory and networking cards in the same server. This section reviews the main benefits of resource disaggregation, which are independent resource scaling, higher resource usage, isolated fault tolerance domains, reduced data migration cost and the possibility of using larger capacities.

1.4.1.1 Independent capacity scaling

One of the promise of resource disaggregation is to offer *independent capacity scaling*. Current server architectures aggregating all resources together needed to upgrade the entire server and all its associated resources to upgrade a single resource. For instance, Storage resources can also be upgraded independently by aggregating storage servers in storage pools handling the logics for growing, shrinking, replicating ....

1.4.1.2 Higher resource usage rate

Storage disaggregation helps to achieve *higher resource usage* by avoiding local servers over-provisioning. Indeed, disaggregated architecture can hot-add and hot-remove resources during the execution by leveraging pool abstractions. For example, storage disaggregation supports expansion of storage capacity at runtime. Thus, it is

possible to register a new filesystem mount point or to attach a new remote storage device during the execution without shutting down the machine.

1.4.1.3 Isolated fault tolerance domains

Servers using attached storage provide fault tolerance by using block redundancy. However, because storage and other hardware resources (CPU, memory, networking) are collocated, a VM can be made unavailable if one of the other hardware resources is faulted.

1.4.1.4 Lower data migration cost

Separating resources enables to easily attach and detach resources to avoid data migrations. It is highly desirable for live virtual machine migration by avoiding data copy and preferring reference copy. Storage disaggregation has already proven it can reduce virtual machines migration time by reattaching source node storage on the destination node.

1.4.1.5 Larger capacities

Disaggregation solutions often target very large memory capacities in a scale-up approach. The Machine from HP Labs ™[83, 142] attaches processors of multiple nodes to a large fabric-attached shared memory pool of 160 TB of DRAM.

1.4.2 The existing storage disaggregation

The idea of aggregating pools of resources on a remote shared instance has already been exploited for storage with distributed file systems and disaggregated block storage. In the following paragraphs, we detail the techniques proposed by disaggregated storage.

1.4.2.1 distributed file systems

First proposals for storage disaggregation proposed filesystem abstraction over a network fabric. Early proposals are NFS [109], which allows multiple client machines to mount a filesystem and perform POSIX-like operations on files. Newer projects have emerged to offer higher bandwidth and leverage existing RDMA fabrics like Lustre [101] or NFSv4 [63].

Filesystems provide easy and standard abstractions however it provides a large set of *metadata services* linked to dentry and inode abstractions (naming, access control, ...). These metadata services, which mostly comply with POSIX filesystem standard, require multiple round time trip to perform simple IO operations. Similarly, distributed file systems offers *data services* like journalling and structured block layout. Some use cases like storing virtual machines images do not need this cumbersome layer.

1.4.2.2 disaggregated block storage

Distributed file systems expose file system interfaces which are more or less POSIX compliant. However, relying on file system interfaces is not always relevant and some use cases require a simpler block interface. One of the reason to rely on disaggregated block storage is that file system interfaces usually require multiple additional operations which are not required with block level storage. Another reason to use disaggregated block storage is that the client-side of distributed file systems requires more logic than block layer storage. Thus, it is more challenging to support distributed file system clients in server firmware prior to OS boot. This causes a bootstrap problem to read operating system binary and makes disaggregated block storage an interesting solution for an operating system working on fully disaggregated block storage.

Multiple block abstractions have been proposed to perform disaggregated storage accesses. In the 2000s, iSCSI [169] was first proposed by porting SCSI protocol encapsulation over the internet. The protocol has later been extended to work with a RDMA fabric known as iSCSI Extensions for RDMA (iSER) [73]. Newer proposals have emerged recently with NVMeoF [112] which offers the benefit of NVMe parallelism to disaggregated storage.

These proposals offer very raw IO storage mechanisms however additional storage services are desirable in many cases. Additional service include support for filesystem abstraction with support for naming and metadata as in Lustre [101] or HDFS [133]. Other solutions provide fault tolerance like CEPH [159].

1.4.3 Disaggregated Memory

Far memory or disaggregated memory refers to any systems using memory of a remote server. The term is used interchangeably for techniques accessing remote memory using IO semantics [7, 61] (i.e. explicit IO operations) or cache semantics [34, 28, 60] (i.e. CPU load/store). The same terminology has been used by work permitting shared accesses to remote memory [28, 5, 34] and work using partition-like accesses [61, 7]. In particular, over the past few years, multiple hardware vendors have been working together to build hardware interconnects between processors and remote memory. We review these interconnects in more details in section 1.5. In particular, among the diversity of proposals, CXL [34] is emerging as the de-facto standard for disaggregated memory accesses with promises to support remote memory accesses with 300 ns latency and high memory bandwidth. We propose in chapter 4 a thorough review of disaggregated memory literature and we review in the following two paragraphs how disaggregated memory can be used in two different way with rack-scale and resource mutualization architectures.

1.4.3.1 Rack-scale architecture

Resource disaggregation is at the core of rack-scale architecture. Rack-scale architecture [71] is a proposal to have servers with highly heterogeneous resources like The Machine from HP Labs ™. Rack-scale servers offer a resource in larger capacity than classic commodity servers to permit independent resource scaling.

However, rack-scale architecture requires a massive shift in server hardware and it has not seen production adoption so far. Physical limitations may also appear to build large specialized servers. For instance regarding compute blades, it may be hard to achieve cooling on a dense set of compute units. Concerning memory blades, there is a *memory density limit* because memory relies on capacitors which have not shrinked for ten years contrarily to transistors. Memory blades may also offer a reduced set of compute units, but processors have a limited number of memory channels which may still require a dependent scaling of compute units with DIMMs. It is not clear yet how servers will deal with these limitations, but more feasible architecture is to mutualize resources across the rack.

1.4.3.2 Resource mutualization architecture

Some proposals [95] build on top of commodity servers and propose an approach around resource mutualization where some dedicated resources of different servers are aggregated into dynamic pools. This approach particularly applies to memory disaggregation. This approach can be seen as the overall memory usage across servers by aggregating leftovers. While early prototypes left memory sharing feature aside it is a desirable feature for such architecture to reduce potential fragmentation and avoid partition based schemes. It appears, shared memory can also help to implement rack-scale communications across servers. The advantage of this architecture is to maintain current local memory bandwidth and latency by avoiding the cost of systematic fabric-attached memory accesses. This approach remains compatible with rack-scale architectures. It has mostly focused industry efforts on building new low-latency high-bandwidth interconnects for various set of accelerators (GPGPU, TPU, smartNICs, ...) while extending memory capacities.

1.5 Cache coherent interconnects

As detailed in previous sections, there exist a large diversity of memory backends and techniques to mutualize rack memory. More and more processors (SmartNIC, CPUs, GPUs, TPUs, ...) and caching devices want to uniformly share access to these memory backends. In order to maintain high bandwidth and low latency whatever the backend, these processors usually rely on hardware caches. However, this ideal configuration where processors share access to a memory region through a shared memory region causes a well-known problem of incoherent memory where processor caches observe different values for a same cache line.

In order to solve this incoherent view, hardware vendors have proposed processor interconnects to implement a transparent algorithm between CPU caches to maintain a coherent view of memory between processors. Since the early 2000s, CPU vendors have been working on improving bandwidth, latency overhead and scalability of these interconnects for multiprocessor architectures (SMP or NUMA). More recently, multiple research prototypes and hardware vendors have worked towards building off-chip cache-coherent interconnects to interconnect multiple devices inside a server but also between different servers. Proposed solutions leverage programmable switches [156, 90] or cache-coherent hardware interconnects [28, 156]. These prototypes are re-

visiting a large part of the literature on interconnect networks with new scalability challenges.

In our context of disaggregated memory, cache coherent interconnects enable multiple compute nodes to perform ***transparent*** *and* ***coherent*** *remote memory accesses on a shared segment. In this section, we discuss the historical bus abstraction before defining interconnect networks and their usage in modern computers. In particular, we focus on existing ccNUMA interprocessor interconnects and next-generation host to device cache coherent interconnects.*

1.5.1 Interconnection networks: from the bus to the interconnect

In the following paragraphs, we review the existing communication primitives used in interconnect networks and the different challenges and trade-off in the design of these primitives.

1.5.1.1 Challenges and solutions of the historical bus abstraction

Before the 1980s, computers relied heavily on a simple interconnect abstraction named *buses*. A bus connects one or multiple modules (processors and memory) on a single shared channel providing *broadcast* semantics. [42] Broadcast means that every message is sent to all the modules which are part of the bus network. Additionally to the broadcast property, buses also implement *message serialization* by letting a single module lock the bus until all modules have replied or acknowledged the request. Thus, buses guarantee total order broadcast meaning that all correct nodes receive the same sequence of messages.

When multiple modules (e.g. processor, DMA controllers, memory controllers) are connected to the shared bus, it is possible that multiple modules try to send messages on the bus concurrently. In order to avoid conflicting accesses to the bus, buses implement a mechanism to grant exclusive ownership of the bus to a module to initiate a transaction.

In interconnect network, the assignment of the shared bus to one of the module is named *arbitration*. The modules requesting ownership of the bus are named *requesters* and the component responsible for granting the ownership to one of the module participating in the bus communications for a lapse of time is named *arbiter*. There exists different architectures for arbitration such a centralized arbitration with a central arbiter (usually the bus controller) or distributed arbitration where a bus master is elected at the beginning of new epochs. Once it has received arbitration, a module becomes *bus master* and can initiate a transaction.

Buses are a simple communication abstraction but they help understand some of the ordering problems relevant to implement cache coherency protocols. The design space of centralized or distributed bus arbitration and the implementation of exclusive ownership also helps to understand in-network cache coherency. Typically, bus broadcast communications enable to implement basic cache coherency protocols thanks to by-design traffic snooping. However, buses also suffer from bad performances because they are serial communication medium by design and they do not

operate at high speed [42]. This has led to the advent of switched interconnects with unicast communication to provide performance, cost reduction and scalability [42] which are described in the next section.

1.5.1.2 Interconnection networks usage in modern computers

An interconnect or interconnection network is a programmable system that transports data between participants. [42] Contrarily to a bus, an interconnect allows *point-to-point interconnection* by relying on switches. Point-to-point communications reduce the number of messages and enable higher-parallelism by avoiding the global bus lock. Thus, interconnects have been widely adopted because they offer better *scaling*. A widely used interconnect is Peripheral Component Interconnect Express (PCIe).

1.5.2 Computer interconnect

In §1.5.1, we have reviewed, bus and interconnects, the main abstractions used in computer interconnects. In the following paragraphs, we discuss how computer maintains a simple load/store abstraction through the use of multiple interconnects in modern and future computers.

We present the main historical and current solutions used from the legacy system bus, to the scalable interprocessor interconnects for ccNUMA before a large overview of next-generation inter-server interconnects.

1.5.2.1 System Bus

From 1995 to 2008, CPU vendors (Intel, AMD) have relied on the *Front-Side Bus* (FSB) also named *System Bus* to interconnect the processor with the northbridge where the memory controller was located. It was identified as a major bottleneck for memory operations resulting in processor being frequently idle. This was largely caused by bus networking which required single memory operations at a time while processor were supporting multiprocessing.

1.5.2.2 Interprocessor Interconnects

In 2001, AMD proposed a more efficient but also more expensive alternative to FSB with the Hypertransport interconnect. In 2009, Intel introduced a new cache coherent interconnect named *QPI 1.0* known at that time as *Common System Interface* (CSI) [82] to support distributed shared memory. QPI has to support a large range of configuration from servers with a few sockets which require high bandwidth and low latency to servers with many processors which require good scaling.

These new interprocessor interconnects rely on *reliable message passing* to communicate between each others. [10] Messages are encapsulated in a layered protocol with physical, link, routing, transport and protocol layers each implementing different services for communications.

Hardware cache coherency protocols differ across interconnects. Historically, processor implemented MESI cache coherency protocol on the FSB bus. However, since MESI protocol generates undesired traffic when a shared cache line is requested with all processors responding to the broadcast request, new protocols have been adopted on interconnects.

In QPI 1.0, Intel introduced source snooping MESIF by introducing a new F state for *Forwarding*. [82] When a cache line is in F state it must invalidate other copies to be allowed to write.
Concurrently, AMD hypertransport interconnect has relied on home snooping MOESI cache coherency with the introduction of the O state for *Ownership*. We review some of the existing protocols in later sections.

1.5.2.3 Rack interconnects

Over the last few years, multiple cache coherent interconnects have been proposed including Compute Express Link (CXL), IBM Coherent Accelerator Interface (CAPI), Cache Coherence Interconnect for Accelerators (CCIX), AMD Infinity Fabric, NVIDIA NVLink. The development of these new cache coherent interconnects is motivated by new workloads notably due to Artificial Intelligence. However, some standards, such as CXL or CCIX, have made room for other use cases which may benefit from a cache coherent interconnect such as SmartNICs or Memory extension cards. All these interconnects face similar problems. One of this problem is that the support of multiple ISA which is hard because there exist different cache coherency protocol implementation, which require dedicated logic and vendor modifications to processors. [140] Another problem is to scale off-chip coherency protocols [140].

CAPI is a standard designed by IBM to attach accelerators (FPGA, GPU) to the host system through a cache coherent interconnect. The first implementation were built on top of PCIe protocol. An extension of CAPI known as OpenCAPI which supports multiple accelerator configurations and memory expansion has been proposed in a standard. OpenCAPI is not based on PCIe even though it shares multiple similarities. IBM released a version of OpenCAPI in Power9 processor making it the first commercially available processor to offer support for hardware disaggregated memory through the Open Memory Interface (OMI) to attach DRAM to OpenCAPI bus. OpenCAPI ended up merged with the CXL standard in 2022.

CCIX [141] is an open-standard standard created to maintain accelerators and processors in the same cache coherency domain. It aimed at supporting multiple ISA. It is built as an extension to PCIe protocol. Similarly to CXL, CCIX supports three different devices: accelerators, accelerators with memory and expansion memory. CCIX defines two components: CCIX agents (devices) and host systems. As stated in the specification [141], CCIX requires changes to the host system to route memory reads and writes to CCIX agents. Moreover, processor caches must respond to snoop requests from CCIX agents. It seems that it did not receive update since 2021 as it is challenged by CXL consortium.

NVLink is a proprietary cache coherent interconnect designed by Nvidia for cache coherent shared memory between GPUs and GPUs and theoretically between CPUs and GPUs. Historically, NVIDIA has provided a *Unified Virtual Addressing* (UVA) abstraction for direct GPU-to-GPU transfers implemented on top of PCIe.

However, PCIe has been identified as a bottleneck for large data transfers between GPUs thus NVIDIA has introduces NVLink for GPU-GPU communications. Thus, NVlink is one of the interconnect which does not rely on PCIe protocol and it actually claims higher throughput although PCIe throughput varies a lot depending on the generation. Contrarily to the other interconnects, NVLink has been designed with a focus on GPU accelerator and it does not target memory extension cards or SmartNICs such as CXL.

CXL is an open-source industry standard. It extends PCIe protocol to provide a cache coherent interconnect for various use cases. It has emerged as the de-facto standard among other open-source proposals, which also targeted multiple instruction set architecture (ISA). We review CXL interconnect in more details in next sections as it appears as the next-generation standard for cache coherent interconnects.

1.5.3 CXL, the next-generation cache-coherent rack-scale interconnect

In §1.5.2.3, we have reviewed various rack interconnects for cache coherent accesses. In this section, we have shown that CXL is left as the de-facto standard among a larger set of interconnects. This section presents CXL in more details with a review of early prototypes and early order of magnitudes for these prototypes.

1.5.3.1 Early Prototypes

DirectCXL [60] is the first CXL.mem research prototype published based on CXL 2.0. It uses a 16 nm FPGA which implements a *CXL controller* to manage CXL endpoints and a *DRAM controller* to attach 8 DRAM modules of 64 GiB. They build their own RISC-V 4 core processors to implement a CXL root port in the Last Level Cache (LLC). In their CPU, they use 16 Miss Status Hold Registers (MSHR), a hardware structure to track LLC misses and exhibit CPU stalling when a cache access spans on more than 16 cache blocks.

Similarly to PCIe, host CPU will have one or multiple CXL root ports which initiates a CXL topology. Topology is composed of *CXL switch* with a downstream port (DSP) and upstream port (USP) and *CXL endpoints* where devices are connected. Root port location and address space is determined during *CXL enumeration*.

1.5.3.2 CXL performances

DirectCXL [60] reports 328 cycles average latency for 64B random access against 2129 average latency for Connect X-3. DirectCXL shows that CXL is close to local DRAM latency for many applications because it benefits from going through CPU caches. They also show that CXL latency is steady compared to RDMA which has large distributions because of memory translation caching.

Pond [95] estimates that CXL memory accesses on memory located in the same NUMA node with 8 to 16 sockets will cost 70 to 90 additional nanoseconds to memory accesses. They also estimate that rack scale memory accesses will add around 180 nanoseconds overhead.

1.5.4 Defining cache coherency

We now review how cache coherency is implemented in general, and then we focus on how it is implemented in rack-scale interconnects. In a system with incoherent caches, a processor would locally mutate a cache block in a few CPU cycles without immediate propagation to shared memory since this would cost thousands of CPU cycles. The mutation without any coordination would remain local with no visibility for other processors. A cache coherency protocol aims at tackling this issue.

Cache coherency can be defined using two invariants:

- **Invariant 1 - Single Writer Multiple Reader**: "For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it." [137]
- **Invariant 2 - Data Value Invariant**: "The value of a memory location at the start of an epoch is the same as the value of the memory location at the end of its last read–write epoch." [137]

A coherence transaction is a distributed transaction which requires serializing conflicting transactions.

1.5.5 Cache coherency protocols

After defining cache coherency, we review the main ideas used to design cache coherency protocols. First, we review some of most common states used to maintain cache line coherency with a state machine. Then, we discuss two of the main architectures which are snoop-based protocols and directory-based protocols.

1.5.5.1 Cache coherency state machine

Cache coherency protocols are commonly represented as a finite state machine to capture states and transitions between them. Common states include: *Modified* (M) to indicate that the cache block can be read or written by the core. *Shared* (S) to indicate that one or multiple cores can perform read-only accesses. *Invalid* (I) indicates that the block is not in a cache or not in a valid state, so no operation may be performed on the block. *Owned* (O) indicates that the block is owned by a core, but other caches have a copy too. *Exclusive* (E) is an optimization state used when a CPU issues a load but is the only reader of the block. This state enables silent transition to Modified state with no additional traffic.

For example, MESI protocol is widely used on multiprocessor cores (SMP) architectures and is usually extended either with *Owned* state (MOESI) or with *Forward* state (MESIF) as in Intel QPI [10]. Different cache coherency protocols may also be used at difference cache levels.

1.5.5.2 Bus Snooping protocols

Snooping-based cache coherency works by having each cache controller use broadcast requests and get unicast response. Broadcast requests are serialized on the

network usually thanks to bus arbitration which enable *total order broadcast* guarantees [137]. Total order broadcast offers convenient properties like serialization of coherency transactions and enables each cache controller to receive the same message in the same order on all cache controllers. Then cache controllers are responsible to implement the protocol. In snooping protocols, a block state is distributed across all cache controllers [137] Thus, when a processor issues access to a cache line, the state of the cache line is known by all processors. However, total order broadcast scales poorly with the number of node and this result in snooping-based protocol to be progressively abandoned in favour of directory-based implementations.

1.5.5.3 Directory Based protocols

Directory-based cache coherency protocols on the contrary of snooping-based cache coherency issues unicast transactions to the cache controller which is the *home* of the block. The home is responsible for looking through an index of cache blocks known as the *directory* to determine the owner of the requests block, its state and cores sharing the block. In directory-based protocols, block state is not maintained on each cache controller which is the main difference with snooping protocols [137]. Directory cache coherency protocols do not benefit from message ordering as in snooping protocols. However it still need to serialize two concurrent unicast transactions on a same block. In directory cache coherency, coherence transactions are ordered at the directory. In case of a race, one of the transaction aborts.

Directory based cache coherency generates less coherence traffic and thus offer more scalability. It is already used in newer interconnect like UPI and provide interesting benefit for rack-scale coherency [156]. A known inconvenient of directory-based protocols is to require an additional message when the home is not the owner of the block. [137]

1.5.6 DMA cache coherency

DMA controller may issue read and write operation to main memory. DMA controller expects to read last version of data but if a processor concurrently work on the same data it may hold it in a CPU cache in Modified state. Then DMA controller must issue invalidation operation to read last version of the block.

x86 supports cache coherent DMA operations, but some architectures do not support it and require explicit arbitration by the driver to let either the device or the host access memory (dma_sync_single_for_cpu() and dma_sync_single_for_device()). This is similar to CXL coherence biases. (see §1.5.7.1)

1.5.7 Rack scale cache coherency

For a long period, different assumptions have been made over the coherency guarantees of disaggregated memory. Some prototypes [155] made the assumption that disaggregated memory would let a server access another server memory in a *mutualization* configuration. In this scenario, the memory server may access the given memory buffer concurrently to other servers. In such a scenario, it is important

that the server accessing remote memory and the server lending its memory are part of the same cache coherency domain.

Other prototypes have assumed that disaggregated memory meant full delegation of physical memory to one or multiple external server. This configuration means that the server lending its memory (compute node or type 3 CXL device) is not expected to access its memory. In such a scenario, cache coherency may still be required when multiple servers would like to concurrently access this memory buffer.

1.5.7.1 CXL cache coherency: coherency bias and enhanced coherency

Because of the problem to scale cache coherency to multiple hosts, CXL rely on a different scheme to ensure cache coherency.

CXL uses *biases* [34] which give hints about which component should pay the cost of cache coherency. CXL proposes to rely on two biases: *Host bias* and *Device bias*. In device bias, pages are only used by the device and the device is ensured that the host has not cached the pages which are accessed. In host bias, pages may be used by the host exclusively or by both device and host.

In CXL 3.0, coherency bias technique has been replaced by a new coherency method named *Enhanced coherency* which is more transparent for the programmer. Enhanced coherency relies on the ability of the device to use *back invalidation* to invalidate a cache line in the host cache. It also supports, back invalidation in other devices connected to the CXL interconnects which means that cache coherency can be maintained with peer-to-peer communications which avoids going through the host.

1.5.7.2 Coherency Domains

For some time, disaggregated memory has been ambiguous in the assumption of supporting cache coherent accesses. Some works [53] introduced the idea of *coherency domains* as a set of nodes participating in the same cache coherency protocol. The idea behind coherency domains was to restrict cache coherency to a limited set of nodes to enable better cache coherency scaling (even if directory-based solution already improve scalability). Coherency domains posed interesting perspective where software would be responsible to reconfigure these dynamically coherency domains for specific use case depending on the workload degree of sharing.

In this section, we have reviewed recent changes in hardware solutions to store data and in interconnects. In particular, we have proposed an overview of heterogeneity by characterizing backends according to a larger set of properties than the conventional view based on latency and bandwidth differences. We have also started reviewing some of the challenges posed by sharing memory across multiple servers with a review of existing and upcoming interconnect cache coherency protocols. In the next sections, we will focus on the different abstractions proposed by operating systems to use memory and newer proposals to manage heterogeneity.

2 Principles of Linux kernel memory management

In the previous section, we have reviewed recent changes to hardware memory backends and we have tried to present key properties of memory hardware. We have also detailed some of the challenges to design usable and efficient memory hardware for software. In this section, we review key mechanisms used in Linux kernel v5.11.1 operating system. The review of these mechanisms is necessary to understand recent software proposals for heterogeneous memory discussed in chapter 3 and to understand our contribution presented in chapter 6 and chapter 8.

2.1 Core abstractions of Memory Management

In this section, we perform a bottom-up review of Linux abstractions used to build memory management. We review software-hardware interfaces such as node abstraction, memory zones before presenting pages for address translation and kernel memory services. Then, we present process abstractions with process address space and process mappings.

2.1.1 Nodes

At the bottom of the Linux memory management stack, there exists a *node* (pg_data_t) for each NUMA node in the system. It contains a simple integer indicating the NUMA node and a list of zones.

2.1.2 Memory zones

A *zone* (struct zone) contains a list of pages (struct page). Zones are typically bound to a zone type. There exists: ZONE_NORMAL type which contains the largest set of pages. ZONE_MOVABLE which is memory that can be moved for defragmentation. This memory zone is commonly used for memory hotplug and hotunplug. ZONE_DEVICE for adding a page (struct page) for each memory devices (NVDIMM) page and thus support memory management kernel services on these devices. ZONE_DMA, ZONE_DMA32 and ZONE_HIGHMEM targets specific device (and ancient) capabilities which are not relevant for the understanding of our

work. Each zone has its own *page allocator* and carry *watermarks* used as threshold for memory reclamation.

A zone groups memory in *pageblocks* containing 2^9 pages (2MiB in x86). Pageblocks tries to group physical pages by migration type. The most important migration types (migratetype) are UNMOVABLE, MOVABLE and RECLAIMABLE (section 2.4). Grouping pages by migration types aims at organizing the physical address space in pageblocks which are either friendly or unfriendly to compaction or defragmentation. It avoids interleaving pages which do not support migration with pages which support it.

Most memory has a MOVABLE migrate type apart from some core kernel components (memory map) which are marked UNMOVABLE. There exist a page allocator free list for each migrate type which means in summary that there exist a page allocator free list for each zone and for each migrate type. When issuing a page allocation, a flag needs to be provided (known as get-free-page flag (GFP)). The GFP flag is converted to a migrate type and the page allocator tries to allocate the page in the free list which satisfies best the GFP flag. For instance, page allocations of user-space anonymous mappings are movable pages allocated with GFP_HIGHUSER_MOVABLE flag which contain MOVABLE and RECLAIMABLE flag meaning this page likely ends up in movable free list of the buddy allocator.

2.1.3 Page

Modern processors rely on memory pages, fixed size units of memory, to translate virtual addresses to physical addresses. Pages are the smallest unit of memory management in the address translation mechanism.

Each base physical page (4K) is represented by a 64B kernel structure (struct page). A struct page describes a page from OS memory management point of view. However, struct page is never directly used in the configuration of MMU (see §2.1.4). This task is handed to special types (pte_t, pmd_t, pmu_d, pgd_t) which represents the MMU view of a page.

Thus, struct page should be seen as a *companion* structure used in most of the logical memory management algorithms. It is the metadata struct for kernel memory management and IO services. This structure carries atomic metadata flags which builds an IO state machine (PG_uptodate, PG_writeback, PG_dirty) or to provide memory hints (PG_locked, PG_swapbacked), ... A page (struct page) also references the address space it is part of, contains the linked list to implement the LRU.

Various abstractions are built on top of pages such as *anonymous memory* for program allocations, *page cache* acting as an IO cache, *compound pages* to aggregate pages together, *network stack pages* with DMA capabilities, Thus, struct page is a very polymorphic structure although, most pages describe anonymous memory and page cache in common datacenter usages. Linux systematically runs a page replacement algorithm on anonymous and page cache memory by placing OS pages (struct page) in LRU lists (see section 2.5).

2.1.4 Virtual Address Space (mm_struct)

As most operating systems using modern hardware, Linux rely on a *virtual address space* abstraction (mm_struct). Any memory access from processor to memory whether it is privileged or not must be made using a virtual address. This abstraction is defined for each process of the operating system. The virtual address space abstraction helps to implement *isolation* and *memory sharing* between processes. It is also used to facilitate the implementation of process *cloning*. It permits to differentiate memory reservation (allocation) from physical memory consumption.

This abstraction requires fast translation mechanism which acceleration is provided by Memory Management Unit (MMU) in a processor. Translation is achieved through a data-structure named *page table* which per-process entry is loaded in a CPU register (CR3 on Intel processors). The MMU and the operating system share a common assumption about the layout of the translation data-structure. Modern processors rely on a hardware caching translation unit named Translation Lookaside Buffer (TLB) to speed up this translation.

2.1.5 Segment (vma)

Virtual address space discussed in previous paragraphs is made of VMAs. A VMA (struct vm_area_struct) is a contiguous virtual memory range granted through mmap system call. A VMA is bound to a unique virtual address space (struct mm_struct), even for mappings shared between processes.

A VMA also holds the flavour of whether the mapping is file-backed or anonymous, shared or private. It holds a vm_ops vtable to translate memory accesses to IO semantics. File-backed mappings will thus find callbacks to perform IO operations (writeback read page, ...) on filesystem backend. Segment sharing between processes is implemented in the file (struct file) abstraction during mmap(). Shared mappings either explicitly manage a shared file-backed mappings or manage an implicit shmemfs file for anonymous shared mappings. Indeed, shared anonymous memory rely on same a virtual filesystem named shmemfs which is backed by a swap device.

Historically, VMAs were stored in a red-black tree and an additional linked list to tackle the problem of slow tree traversal of red-black tree. Since the maple tree patch set [127] VMAs are now uses a B-tree like layout.

Efficient and scalable VMAs lookup is critical for application performances because it is performed on the page fault handler path. Indeed, page fault handler takes the read lock of mmap_lock semaphore. A VMA cache of most recently used VMA was also used prior to its removal with the introduction of maple tree. Multiple research work have focused on improving the performances of VMAs with new data-structures [32]. Another ongoing approach named speculative fault processing [93] tries a transaction approach by issuing page fault handling and verifying for potential concurrent mmap_lock writers.

Operations on VMAs data-structure of a mm_struct are protected by a mmap_sem or mmap_lock Read-Write semaphore. The critical section is entered with write capability upon creation or deletion of a new memory mapping (insertion/deletion of a VMA is commonly made through mmap/munmap system calls). The critical

section is entered with read-only capability in page fault handling for example to perform VMA walk in search of the backing VMA for the address. [1]

2.2 Address translation

Address translation is the service responsible for translation of virtual addresses to physical addresses. Modern computer architectures rely on a hardware Memory Management Unit (MMU) integrated with the CPU to perform address translation. Linux supports software and hardware address translation for multiple architectures.

2.2.1 Generic address translation

On Linux x86, address translation rely on a shared knowledge of the page table data-structure between hardware MMU and software. Kernel is responsible for writing a correct translation configuration in the page table while the hardware reads the configuration and raises interrupts with error codes upon missing or erroneous configuration. x86 layers the page table on 4 to 5 entry levels represented by the types pgd_t, p4d_t, pud_t, pmd_t, pte_t.

The faulting address is stored in CR2 CPU register while the address to the page table is stored in a CR3 CPU register. A kernel thread running in privileged mode is said to be in *process context* when the CR3 CPU register points to a user-space process page table. Concurrent mutations to the page table are managed using a multiple levels of spinlocks called to ensure better scalability. For concurrent modifications of a PMD, a spinlock ptl is embedded in the *struct page* associated to the PMD the PTE is part of. [2] Ultimately, mutual exclusion is provided by the page_table_lock spinlock in mm_struct.

2.2.2 Linear mapping

The structure used by the kernel for address translation is important for the understanding of some mechanisms such as device hotplug and memory hotplug. Additionally to generic address translation for user-space memory, Linux also uses virtual addresses to read or write kernel objects. This means that the kernel also requires page table mappings to be configured to access internal objects.

In particular, for the management in software of kernel addresses, Linux relies on a special optimized mapping which differs from user-space processes mappings. This mapping aims to speed-up translation of virtual addresses to physical addresses by relying on software offset computation instead of walking the page table. The particular mapping is named *linear mapping* and it maps kernel virtual addresses starting at address V on physical address space starting at address P by applying a fixed offset so that $V = P + offset$ at all times. This known offset between the start of the two address space enables to speed up address translation. In reality, this translation technique differs slightly in configurations where the physical address space is not contiguous but made of several sections of contiguous physical ranges.

[1] No VMA insertion is performed in page fault handling thus read lock is enough

[2] Huge page locking is skipped in our explanation for the sake of simplicity

When a new memory device is hot-plugged, the kernel needs to allocate new OS pages (struct page) to execute memory services and linear mapping need to be expended. We discuss this in more details in the chapter dedicated to ExoVM.

2.3 Memory allocations

Kernel memory usage relies on a wide variety of memory allocators which enable fine-grained memory usage in kernel and process contexts. In this section, we review the different allocators used in Linux with a review of Linux page allocator and kernel object allocator.

2.3.1 Page allocations and buddy allocator

Linux implements a *buddy allocator* [87] for pages allocation. Pages are organized as buckets of 2^{order} pages. Buddy allocator leverages a MAX_ORDER buckets of free_area. Blocks are organized in a binary tree. Each free_area bucket maintains a list of free page blocks of a given order between 0 and MAX_ORDER (Commonly 11). It also maintains a bitmap to determine if blocks are used or free. Internally, Linux has made various modifications to adapt to Linux specificities including allocation in atomic or interruptible context, whether direct reclamation is allowed or not... The page allocator interface is:

```
1 struct page *alloc_pages(gfp_t gfp, unsigned int order)
2 void free_pages(struct page *page, unsigned int order)
```

Page allocation has a fast path which goes through free pages and a slow path which requires compaction or reclamation. Fast path issue the allocation request on a target zone before issuing request on fallback zones. It uses a per-zone freelist (zonelist). The slow path performs direct compaction and falls back to direct reclamation and OOM killer described later.

The free call tries to coalesce pages originating from the same bucket (companion buddies) together [59].

2.3.2 SLAB allocator

Linux uses a SLAB allocator [25] to perform internal kernel allocation based on kmalloc. SLAB allocator offers caches for object of same sizes. By default, slab allocator supports allocation classes with all power of 2 sizes from $8B$ to $8kiB$. Linux also supports adding specific SLAB class for object of specific sizes heavily used like inodes, dentry and many other data-structures.

SLAB allocators have two advantages: First, it helps to reduce fragmentation in internal kernel memory management. Second, SLAB allocator helps to speed up allocation of resources on the critical path with per slab locking instead of global allocator locking.

2.4 Principles of memory reclamation

In previous sections, we have seen the core abstractions of memory management in Linux and available memory allocators. In this section, we review how many Linux services hook on memory allocators, especially page allocator to implement kernel memory services. Thus, we first review how the kernel hooks on page allocator before presenting the memory reclamation to free resources and shrink caches upon pressure.

2.4.1 Hooking on page allocation

Memory reclamation is issued following a call to a page allocation alloc_pages or one of its derivative. *Page reclamation* designates the set of mechanisms used by the kernel to retrieve memory.

2.4.2 Background reclamation and direct reclamation

Linux distinguishes two different context for page reclamation: *background reclamation* (kswapd) and *direct reclamation* (try_to_free_pages). Background reclamation tries to maintain a set of free pages for future calls to alloc_page(). It relies on a dedicated kernel thread (kthread) named kswapd which is woken up when there is memory pressure and previous memory reclamation mechanisms failed.

Direct reclamation happens directly in the context of a failure in page allocation. It is thus common that such a failure occurs in the page fault handling. Direct reclamation should be avoided as much as possible since it slows down significantly page fault handling which blocks further progress for the application thread.

Background and direct reclaim eventually share the same code path which performs a layered reclamation.

2.4.3 Layered reclamation

In Linux, page reclamation adopts a hierarchical methodology with decreasing speed to free pages. First, it begins by shrinking preallocated buffer pools like *SLAB caches* (inode, dentry and other object caches). Second, it tries to free pages of file-backed process mappings contained in the *page cache*. Page cache is a good second choice since it is composed of a quickly reclaimable subset of pages (clean pages) and a second slowly reclaimable subset (dirty pages). Third, page reclamation triggers a mechanism named *swapping* which targets anonymous process mappings. It will offload a selected set of pages to be evicted on a backend using IO semantics. Finally, if all previous mechanisms fail to make room for pages it resorts to calling a mechanism named *OOM killer* which will destroy lower priority processes to retrieve memory.

2.4.3.1 SLAB reclamation

To speed-up SLAB allocation, the kernel maintains small pools of pre-allocated objects. These objects may be reclaimed using shrinkers, a generic interface where

clients (e.g. most file systems, KVM MMU, balloon driver) register callback method to count the number of objects to be reclaimed and a method to actually reclaim these objects. The shrinker threads maintains a control structure with the desired number of pages it wants to free. SLAB reclamation may not always be efficient because of the fragmentation objects cause on pages.

2.4.3.2 Page Cache reclamation and Swap reclamation

Page cache and swap reclamation shares a lot of memory management mechanism for reclamation. They rely on the same LRU code paths which are discussed in section 2.5. However, page cache performs eviction of user-space file-backed pages only while swap performs eviction of user-space anonymous pages.

A key difference between page cache reclamation and swap reclamation is how pages are mapped to blocks. Indeed, page cache acts like a real cache, since it always has an allocated page on the storage backend to evict its page. However, swap is different because it supports registration of smaller storage backend than the sum of anonymous memory used in the kernel. Thus swap requires allocation and mapping of pages to the underlying storage backend by storing an encoding of block device ID and offset swp_t in the page table entry.

Another difference is that clean file backed pages are always coherent with the storage backend in page cache. In swap system, there is no guarantee that a clean page in memory exists on the storage backend.

As described previously, page cache reclamation is very quick when page cache owns a significant number of clean pages i.e. pages with cleared PG_dirty. These pages are just dropped since their content already exist on the underlying backend. However, pages which are dirty, i.e. with PG_dirty bit set in struct page require writeback on the underlying backend.

2.4.3.3 OOM Killer

When running out physical memory, Linux Kernel provides an ultimate solution killing a process to reclaim its memory. OOM killer is called in page allocation slow path after failing at *direct reclamation* and *direct compaction*.

Process selection is issued in select_bad_process(). It selects processes using large amount of memory (oom_badness()) by evaluating the task Resident Set Size, page table size and swap space used.

There exist mechanisms to avoid OOM killer to be triggered by performing checks on virtual memory management. Typically, when virtual memory mappings resize themselves with brk() or mremap() calls for instance, they can create memory overcommitment scenario. This is provided by OVERCOMMIT_NEVER policy to avoid virtual memory to create scenario which are likely to lead to overcommitment. However, this requires user space applications to perform appropriate error handling. The default Linux policy is OVERCOMMIT_GUESS which tries to assess upon virtual mappings resizing whether available memory (System RAM and Swap space) may lead to overcommitment scenario.

2.5 LRU based reclamation: Page Cache and Swapping

In previous section, we have present an overview of memory reclamation in Linux kernel with a review of layered reclamation with page cache and swap reclamation. But, page cache and swap reclamation try to perform smart selection of pages to evict to limit performance degradation. In this section, we present how page cache reclamation and swap share a common Least-Recently-Used (LRU) implementation to isolate cold and hot pages. Thus, we first present the design of the LRU algorithm in Linux v5.11.1. Then, we describe how page IO works for swap and page cache with a focus on swap with swap cache, backend sector allocation. Then, we quickly detail page table entries management for swap. Finally, we review page IO abstractions to synchronize pages on a backend.

2.5.1 Design of the LRU page replacement algorithm

Linux uses a variant of Least Recently Used (LRU) algorithm. The goal of this algorithm is to select for eviction the set of pages that has not been used for the longest time.

2.5.1.1 Access bit setting

Linux tries to maintain a list of accessed pages (named *referenced pages* in Linux parlance). Accessed pages are tracked by both software and hardware constructs by setting a page flag PageReferenced in struct page flags. In Linux v5.11.1, Linux looks for accessed pages by scanning the list of pages (struct page) in physical memory.

Hardware is used after each processor access to a mapped page, the hardware MMU sets *access* bits on page table entries (pte_t) which have been accessed. MMU access bit setting is fast but scanning accessed pages in software requires longer time. The access bit in pte_t is made visible to the page control struct (struct page) by setting PageReferenced by calling page_referenced. This method leverages a walk of reverse page mapping (struct rmap) to find all page table entry referencing the page and returns whether the page is marked as accessed or not (page_referenced_one()).

Software is leveraged to set the PageReferenced bit flag in struct page explicitly through mark_page_accessed for page accesses performed with no involvement of the MMU like for file accesses (read/write system call) or when pages are swapped in, calls to get user pages, unmapping of VMAs.

The PageReferenced bit is cleared after every call to page_referenced.

The ideal LRU where every access to a page would result in reordering pages in the list is too expensive in practice [16]. Thus, Linux relies on periodic checks performed in *reclaim paths only* and verifies accesses since last check.

2.5.1.2 Dirty Bit setting

Similarly to access bit, determining if a page has been modified is performed by both software and MMU. Thus, Linux also tries to perform periodic monitoring

of MMU pages (pte_t) to report dirtiness in OS pages (struct page). Linux performs this update from MMU pages to OS pages when the page is unmapped from all process page tables. The reporting procedure first retrieves the old page table entry value (ptep_clear_flush()) before removing the page from the page table and invalidating the TLB.

2.5.1.3 Balancing the active and inactive lists

Linux appends all system pages (struct page) apart from SLAB pages to LRU queues (struct list_lru) because it can not run simply with SLAB objects evicted on non-byte addressable backend. To detail the LRU algorithm, let us consider that Linux relies on two queues: active and inactive (enum lru_list). Operations on LRU list are protected by the *LRU lock*, a spinlock protecting both active and inactive list.

Reclaim threads runs through all LRU lists and try to shrink them with a page budget. It evaluates page_referenced and mark_page_accessed at every round for each list to determine the action appropriate action to perform. Operations possible on the pair of LRU queues are insertion, eviction, demotion, promotion. *Insertion* of a new page is performed to a page to the LRU inactive list. *Eviction* of a page is performed when a page is removed from inactive list and written back on a backend. *Promotion* moves a page in the inactive list to the active list while *demotion* does the reverse operation. Eviction, Demotion, Promotion are performed by the reclaim thread in the call graph of shrink_node.

Only a subset of each LRU list is scanned starting from the tail of the list.

2.5.1.3.1 Page Insertion. Most pages are enqueued at the head of the inactive list and marked with PageLRU flag upon first access. This flag is used to indicate the presence of the page in any of the LRU list (active or inactive). When a new page is inserted in a LRU lists, the tail page is dequeued from the inactive list to be evicted [16]. To speed-up operations and reduce lock contention, Linux actually leverages a per CPU *LRU cache* to batch addition of pages to the inactive list. Most pages are enqueued in inactive but there exists exception like pages read from swap cache which are directly enqueued in active list.

2.5.1.3.2 Page Demotion. Moving least used pages from active list to inactive list is called *demotion* and it is performed in shrink_active_list(). Pages are moved from the tail of the active list to be inserted at the head of the active list in order to maintain ageing order in the queues. It occurs when a previous call to page_referenced() already cleaned the PG_Referenced bit and the page is in active list.

2.5.1.3.3 Page Promotion. *Promotion* describes the operation where a page is removed from inactive list to be place in active list. A page which has already been marked referenced and which is in the inactive list is promoted to the active list. Promotion is performed in mark_page_accessed() which calls activate_page() if PG_referenced bit is already set.

2.5.1.3.4 Page eviction. The actual page reclamation occurs in shrink_inactive_list () after shrinking the active list. The procedure selects a set of pages in inactive list and call shrink_page_list () There are different cases to consider at this point.

First, for all anonymous pages (PG_anon & PG_swapbacked) which are not in swap cache, an allocation on the swap backend is performed (add_to_swap() as described in §2.5.3. The allocation also sets the OS page (struct page) as dirty.

Second, If the page is mapped in the page table of one or more processes, this method tries to unmap it in the page tables (try_to_unmap()). Unmapping of the MMU page in page tables is performed by running a reverse map walk during which a dirtied MMU page (pte_t) causes dirtying of its associated OS page (struct page). The location of the page on the swap backend is encoded in the MMU page (pte_t) during this reverse walk.

Third, if an OS page is dirty, shrink_page_list () will perform TLB flush and writeback of the page on the backend (pageout()). During writeback, the LRU subsystem sets the page flag to PageReclaim and invoke the IO subsystem to writeback (PageWriteback) the page.

Fourth, if the page is clean (PG_dirty bit cleared) and if it already exists on the backend, then the page is discarded (try_to_release_page()).

Fifth, Pages which have been marked as lazy free by a madvise system call with flags MADV_FREE are flagged PG_Anon && !PG_swapbacked and can be dropped immediately.

If the page has filesystem metadata, kernel tries to clear the metadata and to free the page.

2.5.1.4 LRU lists

Real implementation of LRU algorithm uses multiple LRU lists. First, there exists LRU lists for active and inactive pages as seen previously. Second, file-backed pages are appended to *file-backed lists* and anonymous pages to *anonymous lists*. Third, memory cgroup, which enables control of physical memory used by a set of processes, requires per-cgroup LRU lists. Fourth, LRU lists are per NUMA node. The cartesian product of active and inactive pages with file-backed and anonymous lists is called a lruvec struct lruvec. A lruvec describes the entire logic of the LRU algorithms the remaining lists are used for isolation purposes and NUMA performances.

2.5.1.5 Reclamation start and stop: Watermarks

As seen previously, reclamation may be initiated as a background task instead of waiting for critical page usage to occur. One of the challenge is when to wake up background reclamation task. Linux relies on customizable watermarks in calls to alloc_page() to determine when background thread is woken up and which action to perform.

Each kernel zone contains NR_WMARK watermarks. Currently, there are four watermarks defined: WMARK_MIN, WMARK_LOW, WMARK_HIGH, WMARK_PROMO.

```
if zone->free_pages < low_wmark_pages(zone):
        wake up kswapd

if zone->free_pages > high_wmark_pages(zone):
        put kswapd to sleep

if zone->free_pages < min_wmark_pages(zone):
        use direct reclaim
```

2.5.1.6 The contiguous block of memory problem

A known issue of current LRU algorithm is that it favours grouping pages by age rather than by locality [17]. It is known to cause page fragmentation and to complicate the creation of contiguous group of pages which would be useful for buddy allocator, contiguous memory allocator or creation of huge pages to reduce TLB misses.

2.5.2 Swap cache

Linux leverages a writeback cache named *swap cache* placed before the swap backend IO path and which contains pages. Swap cache (struct address_space swapper_space) is a radix-tree like cache indexed by swp_entry_t.

The swap cache is used for pages of a shared anonymous mapping. Multiple page tables have a mapping to an identical physical address which may be evicted in swap-out path with an update in the page table of each process to point to an offset on the swap device (swp_entry_t). To do so, reclamation thread uses reverse mapping to update other processes sharing the mapping.

When one process requires the page back in memory, it either needs to fetch its own version of the page, which would break the shared assumption, or to lookup if other processes have already swapped in the page. Checking whether a page has already been swapped in would require expensive synchronization between processes as well as expensive page table walks. So instead, the process looks up the swap cache first to determine if the page exists and then tries to fetch it.

Thus, Linux adds shared pages (struct page) when the first process that requires the page swap it in. The page is then tagged with PG_SwapCache flag. The page may be removed if the kernel needs to swap-out the page during memory reclamation or if all processes which have a reference on the page have swapped in the page.

2.5.3 Sector allocation and release

Swap mechanisms must also perform reservation of sector on the swap device. Linux divides swap sector space into clusters of 256 pages.

From a high-level perspective, swap supports *sector allocation*, *sector reuse* and *sector discard*.

Sector allocation rely on a *next-fit* allocator with per-cpu allocation caches. Swap system maintains indices for next free cluster in swap space and next free index in

the cluster. A per-page bitmap (swap_map) is used to track usage of sectors and help perform sector discarding.

Sector discarding is also used to reduce wear-leveling on SSD. It is implemented linked list to aggregate discarded clusters until a discard job enqueued in a work queue is scheduled for actual sending of the discard command.

2.5.4 Reading swapped entries

Page table entries propose a present bit (_PAGE_PRESENT) to let the MMU know whether the page can be directly accessed or not. If the page is not marked present, a page fault is generated leading to a swapin operation in the page fault context swapin_readahead(). The operation reads the desired page as well as neighbouring pages. It is motivated by the observation that contiguous pages are usually accessed together.

2.5.5 Page IO

Page IO mostly rely on the use of two OS page flags which are PG_writeback, PG_locked.

The flag PG_writeback indicates that a page is currently under writeback. Linux provides high level API around end_page_writeback() and wait_on_page_writeback() to make processes wait until termination of page writeback.

Linux relies on PG_locked bit to prevent accesses to the page. It is used as way to block concurrent operations on the page. Thus, prior to reading a swap page with swap_readpage(), the page is protected by invocation of __SetPageLocked(). Linux also provides a mutual exclusion abstraction with lock_page() and unlock_page(). This abstraction uses a wait queue to enqueue tasks waiting for the page IO to complete.

In this section, we have proposed a detailed description of memory management in Linux kernel. Changes to kernel memory management remain frequent however the ideas presented here propose a more advanced understanding of design choices in Linux and the services offered by a modern operating system.

3 Heterogeneous memory management

After reviewing in the previous section the principles of general memory management in the Linux kernel, we propose in this section to review the different proposals to handle memory heterogeneity with page placement techniques on multiple tiers. In particular, we begin by discussing available solutions used to report heterogeneity of memory to the operating system. Then, we review memory management mechanisms available to let the end user allocate pages on explicit memory tiers. Finally, we review existing techniques for automatic page placement at kernel level and user-space level.

3.1 Hardware heterogeneity reporting

In systems using different types of memory, the kernel requires finding where each memory device registers in the physical address space and what are the properties or performances of each device. Since, there is no standard regarding physical address space layout, it is important to have a way to programmatically retrieve the differences across these backends through a topology.

3.1.1 ACPI tables: Reporting memory heterogeneity

One of challenge of memory heterogeneity is to let hardware report cost of remote memory accesses.

3.1.1.1 e820

Historically, BIOS boot used e820 to report a memory map to the Operating System or bootloader. The kernel issues a INT 0x15, EAX = 0xE820 command to retrieve memory areas. In Linux kernel, this table is used to report physical memory ranges available as System RAM for page management. It can mostly detail the memory type associated with the physical memory range (RAM usable for OS, persistent memory, disabled memory) [2].

3.1.1.2 SRAT

Since the advent of UEFI firmware (or EFI), a new set of tables has been introduced to report memory map information. ACPI uses System Resource Affinity Table (SRAT) to report processors, memory ranges, accelerators, DMA controllers... In ACPI language, each block of processors and memory deemed close are assigned a number called *Proximity Domain*, which roughly corresponds to a NUMA node.

3.1.1.3 SLIT

ACPI uses another table to report heterogeneity of accesses across NUMA nodes. ACPI defines System Locality Information Table (SLIT), a *locality matrix* representing a graph of NUMA memory node and supported inter-node memory accesses. On this graph, edges weight stands for a normalized memory access latency. However, SLIT reported latency are known to be inaccurate even though reported values are more and more accurate. Moreover, SLIT fails to express performance differences in terms of both latency and bandwidth as it reports a single distance value.

3.1.1.4 HMAT

Since ACPI v6.2, ACPI introduces Heterogeneous Memory Attribute Table (HMAT) [175] to describe memory bandwidth and latency from the point of view of any memory request initiator. It reports three information: Memory Proximity Domains, System Locality with Latency and Bandwidth information, Memory Side Cache information. First, HMAT introduces two sub-classes for memory and processors to SRAT *Proximity Domains*. Thus, *Memory Proximity Domains* describe a set of memory resources while *Processor Proximity Domain* may represent either processor resources or processor and memory resources. Second, HMAT proposes a detailed description of locality with the introduction of a new locality matrix standardized as *System Locality with latency and bandwidth information*. This matrix provides information regarding bandwidth (MB/s) and latency (ps) for read and write operations as well as information regarding hit or miss cost in cache layers. Third, *Memory Side Cache Information* is used to describe caches of platforms supporting caching on the memory side but it does not describe CPU caches (L1, L2, L3). On these platforms, HMAT reports for each memory domain the number of cache levels, their associativity and write policy (writeback, writethrough).

3.1.2 New EFI memory types

The recent arrival of new heterogeneous hardware has led to the adoption of new memory types in the EFI standard. In particular, we present EFI specific purpose memory and a device table to report coherency.

3.1.2.1 EFI specific purpose memory

Since EFI 2.8 [148], a new type of memory named *specific purpose* can be registered. This memory is reported distinctively to ban its usage from classic System Ram usage. One of the use case is to target HBM. Linux registers EFI specific

purpose memory as *soft-reserved memory* which probes a DAX device driver (see §3.2.2). Linux creates dev-dax instances (presented in §3.2.2) for this type of devices which lets the end user the ability to add this memory to the System RAM pool if desired.

3.1.2.2 CDAT

Coherent Device Attribute Table (CDAT) [33] provides coherency information about the devices. CXL devices are expected to report performance metrics using CDAT.

In this section we have reviewed multiple existing and upcoming topology reporting mechanisms. Reported information enable to differentiate the different backends in the physical address space. However, reported information remain mostly inaccurate and there exists multiple different tables to report similar information. Moreover, the set of properties covered by these tables is limited compared to properties reviewed in previous chapter. Indeed, all tables mostly focus on persistence and average bandwidth and latency with no distinction between read and write operations.

Topology reporting remains important for kernel memory management but it seems to be mostly used to draw the border between backends. In next sections, we review the different choices available for memory management on heterogeneous memory.

3.2 Explicit Heterogeneity Management

The design space of memory management of heterogeneous memory is mostly composed of explicit management and automatic management. Since the performance impact of page placement on memory tier depends on various factors which are application specific, an ideal solution to support optimal placement for all type of application is to defer choices of object placement directly to the application. In the following section, we review this technique known as explicit memory management.

3.2.1 System-RAM

System RAM is the set of physical memory resources managed by the kernel. System RAM memory is used by the kernel for its caches and objects as well as for user processes mappings (anonymous or file-backed). Blacklisted ram through specific boot time arguments is not part of system RAM.

3.2.2 DAX, a new mapping type for page cache bypass

DAX is a Linux kernel mechanisms which offers a process real memory semantics by letting it perform direct memory accesses without using main memory.

Initially, DAX has been designed to support NVDIMMs with the support of two important properties which are *page-cache bypass* and *direct memory accesses*. On the one hand, some user-space applications, in particular applications supporting transactions on persistent memory like databases wants to use the persistence guarantees offered by NVDIMM. These applications need to bypass any volatile software cache (e.g. kernel page cache) when they perform memory accesses to benefit from persistence guarantees. Thus, DAX ensures that processes performing memory accesses on NVDIMM bypass the page cache. O, the other hand, Linux filesystem syscall interface also supports I/O operations with page cache bypass with O_DIRECT. However, the O_DIRECT interface is designed to work with read and write system calls to perform file I/Os not for file-backed mappings. Thus, DAX supports direct memory accesses circumventing volatile caches with virtual memory semantics for the program. This is important with NVDIMMs as they work at CPU cache granularity rather than page granularity.

After its success for NVDIMMs, DAX has also become an indispensable mechanism for other new memory backends.

For instance, in processors equipped with HBM, the memory capacity is limited because of the cost per capacity. It is important that performance-critical applications which wants to draw the best from high bandwidth property can use most of HBM memory capacity. Thus, DAX offers a way to prevent the kernel from automatically using HBM and to save the resources for high-performance applications. HBM also benefit from DAX *page-cache bypass* and *direct memory accesses* properties to fully leverage the performance of HBM.

DAX is also useful in the context of remote memory accesses with CXL. Indeed, DAX can be used in this context to expose the heterogeneity of backends and to let applications perform explicit placement of objects in the virtual address space.

Thus, DAX can be considered as a mechanism to bypass *in-kernel automatic page placement* decisions.

3.2.3 NUMA page allocation policies

The physical address space is usually aware of the NUMA memory topology by having separate sections for each NUMA nodes. However, by default, virtual address space in NUMA machines maintains transparency for applications and expose a uniform view of memory. An application requires the use of an explicit system call on a virtual address space range to configure a *NUMA policy* for page allocation during page fault handling.

Linux offers MPOL_LOCAL to perform *first-touch allocation* or *local allocation* by issuing the page allocation on the same NUMA node as the CPU issuing the request. MPOL_INTERLEAVE to perform round-robin allocation on a set of nodes. MPOL_BIND for *strict* allocation pattern on a set of nodes with no allocations outside provided set of nodes. MPOL_PREFERRED to privilege allocations on a single node. MPOL_PREFERRED_MANY to prefer page allocations on a set of nodes and ultimately issuing allocations on the other nodes. This last operation is designed in particular for memory tiering with the possibility to explicitly declare that a page should be allocated either on fast or slow memory tier. The default policy MPOL_DEFAULT is set to be MPOL_LOCAL.

The decision of allocating a page on a local NUMA node or a remote node gets inspired by the page reclamation watermark policy described in chapter 2. That is to say, by default Linux performs local page allocation while zone free pages is greater than low_watermark. When, watermark is crossed background page reclamation on the zone is initiated and page allocator look for free pages on remote NUMA nodes. When the number of free pages in a zone is greater than high_watermark, reclamation can stop and page allocation can resume on local NUMA node.

3.3 Updating the LRU for multiple levels of heterogeneity

In systems using caching layers, the size of the cache is commonly smaller than the size of the backend. Thus, the caching layer requires to update the set of loaded entries and thus may need to determine which entries to evict to make room for newer entries. Cache systems rely on Least-Recently-Used (LRU) algorithms which tries to select a clever set of entries for eviction based on their access pattern. As seen in section 2.5, Linux implements a LRU algorithm to classify pages into two levels of heterogeneity. Indeed, Linux historically required the LRU to implement swap and page cache mechanisms which only considered fast memory (RAM) and slow memory (HDD storage). However, the diversity of memory and storage backends (see chapter 1) now require the LRU to classify pages into more than two levels of heterogeneity to server better page placement.

3.3.1 MGLRU

In section 2.5, we have covered the details of the LRU algorithm in Linux kernel. We have seen, how the current LRU relies on the use of the active and inactive list to determine the set of active pages, inactive pages and evicted pages. Current LRU algorithm has been built under the hypothesis that there were mostly two memory tiers composed of volatile memory and slow storage. Thus, a new proposal, MGLRU, has been made to upgrade the existing LRU algorithm to take into account a broader set of performance among tiered devices.

Multi-generational LRU (MGLRU) [36] (merged in Linux 6.1) proposes to use multiple LRU lists instead of the current two lists. MGLRU maintains lists by age, ranging from the list of older pages to the list of younger pages.

Similarly to legacy LRU, *page promotion* is based on moving referenced pages into younger lists after each scan. Pages are demoted (evict_folios()) after two consecutive scan observed a cleared access bit in the page (pte_t).

Since, swap and page cache mechanisms must be maintained for backward compatibility, MGLRU needs to propose a solution for page eviction. Thus, page reclamation (see section 2.4) select pages to be evicted in older lists instead of picking them at the tail of inactive list.

Instead of scanning all pages in physical memory, MGLRU walks through process page tables. This design choice is motivated by the overhead in legacy LRU which were caused by page lookup in the reverse map (rmap) to look for page table entries (pte_t) referencing the page.

When compared with legacy LRU, MGLRU has shown [134] that it was able to reduce the number of *direct reclamation* and that it was able to reduce *working set refaults* (i.e. bringing back an evicted pages). However direct reclaim latency tends to increase with MGLRU compared to legacy LRU.

MGLRU is a very interesting approach and should pave the way for lots of contributions. Indeed, this new LRU algorithm moves away from the binary approach of legacy LRU which mostly needed to consider whether enough memory remains on the system before resorting to page reclamation mechanisms. Instead, the approach paves the way for careful tuning of page placement on a variety of memory backends by using a new abstraction named generation which groups a set of pages with similar page accesses patterns together. Thus, MGLRU appears as a promising framework to implement custom policies to allocate generations on memory backends and to migrate pages between the backends using generation lists information.

3.4 Remote caching

Instead of relying on local memory to cache information, some early works made the opposite choice of using remote memory for caching. The idea of these work is based on the observation that remote memory accesses may be faster than disk IO which means that remote memory could be used as a cache. Such prototypes proposed since the 1990s account as one of the earliest usage of remote memory along with distributed shared memory prototypes. Many techniques proposed in these works are similar to current disaggregated memory prototypes, however, they remain mostly

focused on caching files contents or compilation results.

3.4.1 Cooperative Caching

Thus, *cooperative caching* [41] proposed to rely on networking capacity to offload file-backed pages to remote nodes. In the original idea of cooperative caching, the idea was also to propose merging of individual local caches into a larger shared page cache for all members of the cluster. They showed that cooperative caching could reduce by 50% the number of disk accesses while improving page read.

More recently, PUMA [100] has proposed a cooperative caching solution for IO intensive virtual machines to achieve higher consolidation ratio in datacenters. PUMA exposes a block device to perform transparent RDMA accesses. PUMA modifies Linux Page Cache to allow remote page invalidation required for distributed file systems as this feature is not available in native Linux page cache.

Based on the observation that there is more than file content to cache, some works [84, 40] propose to cache compiled code in remote memory. Since programs rely on external code provided by libraries, there exists a decent part of code which is common across applications. JITServer [84] and SHMVM [40] propose to reduce JIT compilation overhead by sharing JIT results. It allows for code reuse across datacenter by sharing results of compiled native Java code.

3.5 In-kernel automatic memory placement

As opposed to explicit placement of pages by applications, the kernel may propose a service to automatically infer where pages should be located on heterogeneous memory. There are two main approaches to improve thread locality in NUMA architectures in Linux.

The first approach performs task placement, by migrating a task close to its memory. However, task placement on tiered memory (i.e. NUMA nodes without processors) is not relevant since tiered memory has no processors to execute code. Thus, placement of execution unit and memory servers with tiered memory require an alternative approach.

The second approach performs page placement by migrating pages close to processor performing the access or on a quicker memory backend. This approach works for servers with CPU NUMA nodes or CPU-less NUMA nodes (tiered memory). In this section, we mainly focus on page placement but we also discuss task placement.

The main idea behind page placement is to track page accesses frequency to to classify cold pages set (i.e. lower access frequency) from hot pages set (i.e. higher access frequency). All techniques strive to place the hottest pages on the fastest memory backend and the coldest pages on the slowest memory backend. Once page classification is performed, page placement algorithms rely on page migration mechanism to move pages on the different backends. In this section, we first review page migration mechanisms which have a limited set of differences between each others. Then, we discuss the tracking and classification parts of existing prototypes with an introduction to the classic page table walker and its limits before discussing tracking

and classification on heterogeneous memory made of CPU NUMA nodes (common NUMA) and CPU-less nodes (tiered memory).

3.5.1 Page Migration

Page migration is a key part of automatic page placement since it supports moving a set of pages to another region in physical memory. Page migration is used for various use cases such as handling memory failures, defragmentation of memory during memory hot-unplug, compaction of pages in larger pages (e.g. compaction of 4 kiB pages to 2 MiB pages), or moving misplaced page in NUMA machines.

Linux relies on a single-threaded page migration implemented in the kernel. The method is named migrate_pages() [1]. Various memory management system calls such as mbind(), move_pages(), or migrate_pages() rely on this migration method. The migrate_pages() method supports moving a set of *from* pages to a set of newly allocated *to* pages. The method blocks all accesses and mutation to the source and destination pages. It unmaps all associated PTE (try_to_unmap()) and move the page content. Since multiple services rely on page migration, the page migration interface lets the caller devise whether migration should be a blocking or non-blocking operation. In the context of automatic NUMA placement, page migration is performed asynchronously (NUMA_ASYNC).

There exists alternatives to the serial page migration implementation used in Linux kernel. For instance, Nimble [166] and Autotiering [85] rely on parallel page exchange for Transparent Huge Pages. In this review, we will not dwell on the alternative techniques in the literature for page migration. However, page migration is a slow operation which requires expensive mechanism such as TLB shootdown. It can be even slower when huge pages are used [70]. One of the main issue of page migration is that these techniques cause stop-the-world phases for source and destination pages during the copy.

3.5.2 Page Tracking

Page tracking is a desirable feature for various kernel services. Usually, it tries to determine the set of accessed pages and the set of written pages. For example, it is required to determine *page accesses frequency* for *working set estimation* or to issue optimal placement decisions. *Dirty page tracking* may be used for *checkpoint-restart based* algorithms or for *VM migration* to try to maintain a coherent state between two replicas. A good page tracking solution needs to be accurate and to have a minimal overhead on application execution.

3.5.2.1 The high-overhead of classic page tracking

The **classic page tracking** method used to separate a set of hot pages from cold pages is to perform page table inspection and read Access and Dirty bit in page table entries. When the MMU reads or writes a page table entry, it sets the Access

[1] Here, we refer to the kernel memory management method not to the system call of the same name

bit of this page table entry. For writes, it additionally sets the Dirty bit of the page table entry (PTE). A page table scanner acts as a dedicated thread inspecting page table entries for newly accessed or dirtied entry. It maintains a map of pages access frequency before clearing the access and dirty bits.

However, this classical approach requires *entire page table scans* which is unaffordable with ever-growing memory capacity.

Another problem is that classic page tracking requires a TLB flush after every A/D bit clearing to make sure the MMU will set it again upon future access. This TLB flush leads to remote TLB flush which is an expensive operation. Additionally, this entire TLB invalidation hurts next memory accesses.

3.5.3 Automatic page placement on NUMA nodes with CPUs

AutoNUMA [14, 58] or *Automatic NUMA balancing* is a transparent memory management mechanism which proposes an alternative page tracking solution. It aims at improving page accesses locality in NUMA machines by performing *task migration* and *page migration* to reduce remote memory accesses. On the one hand, task migration tries to migrate threads closer to the pages they access. On the other hand, page migration tries to migrate pages closer to threads that use these pages by following memory policies.

3.5.3.1 NUMA hint fault

AutoNUMA needs to retrieve information about the frequency of page accesses as well as which processor accesses each page. This information is retrieved using a page fault instrumentation method named NUMA hint fault which is used for both *task migration* and *page migration* use cases. The instrumentation method is split into two parts, scanning and faulting.

Scanning could be implemented as a dedicated kernel thread scanning all page tables in the system. However the use of a dedicated kernel thread would mislead the scheduler to grant as much CPU time to the page migration thread as it would give to user-space applications. Thus, this would hurt scheduler fairness. Instead, scanning is implemented within the scheduler code and is executed in each process context. This way the scheduler can ensure that page migration never exceeds a certain limit of CPU time. When invoked by the scheduler, the scanning job clears some bits in the page table entries of a region of process memory (the default region size is 256 MiB). This will make any subsequent access to the page trigger a page fault.

One of the challenge in the implementation of the NUMA hint fault is to find available bits in page table entries to distinguish the NUMA hint page fault from other page fault scenario in the page fault handler code. In particular, the NUMA hint fault needs to be distinguished from three other similar scenario which are page faults for a resident page table entry protected from read-write-execute, for a swap page table entry and for an entry that is unmapped. In x86, Linux must encode the NUMA hint fault by using multiple bits in the page table entry. There has been multiple changes in the encoding of the NUMA hint fault, but since Linux v4.0, the hint fault is encoded with the protection bit cleared in the page table entry.

Faulting occurs when a processor accesses a page which as been hinted during scan time. It causes a special page fault named *NUMA hint fault* (do_numa_page()) which is detected by checking if present bit is cleared in pte_t and global bit is set. The CPU performing the access is recorded in an array of the structure backing the process (struct task_struct) (in task_numa_fault()). A remote access is performed when the processor accessing the page is not part of the same NUMA node as the page.

3.5.3.2 Task migration

During the NUMA hint fault accounting phase (to determine if page accesses are local or remote), AutoNUMA tries to perform *task migration* to move a task as close as possible to NUMA memory. It determines a preferred NUMA node (task_numa_placement()) for the process based on which NUMA node emitted most NUMA hint fault. Then, it performs task migration (numa_migrate_preferred()).

3.5.3.3 Page migration

AutoNUMA is not just about *task placement*, it also performs *page placement* by enforcing memory placement policies defined by the application during page fault. This is commonly named *migrate-on-fault* as during a *NUMA hint fault*, AutoNUMA determines if the faulting page NUMA node matches the per-segment (struct vma_struct) NUMA policy (mpol_misplaced()). It takes into account the NUMA policy and tries to determine a target NUMA node. If a target NUMA node that is different from the requesting CPU is found, page migration is called (migrate_misplaced_page()). However, page migration may fail typically when destination NUMA node does not have enough free pages.

3.5.4 Limits to Automatic page placement on CPU-ful NUMA nodes

The solution offered by AutoNUMA in Linux remains unsatisfactory for many reasons studied through different works in the literature. These works have identified AutoNUMA to cause lots of cache pollution, to require lots of traffic to maintain TLB coherency across cores. Finally, AutoNUMA assumes that NUMA nodes are all comprised of processing units and does not consider the existence of memory-only NUMA nodes.

3.5.4.1 Data cache pollution

One of the reason is that page table walk has been identified to cause lots of data cache pollution (L1, L2, LLC) because it needs to access information of many pages in a segment (struct page) to determine if the page may be migrated [15]. For instance, the page information is used to skip migration of some dirty pages and from copy-on-write mappings.

3.5.4.2 TLB shootdown

On top of data cache pollution, AutoNUMA page migration is also known to cause lots of expensive TLB shootdown [89].

Hardware MMU leverage a small cache to remember page table address translation. This cache is called Translation Lookaside Buffer (TLB). There exist a TLB cache for each processor but a single page table can be shared in main memory by all processors. Thus, TLB caches may be in a incoherent state (see §1.5.4) which means that mapping information may not be identical for a same process in all TLBs. Contrarily to data caches coherency which is maintained by the hardware, TLB caches coherency is delegated to the operating system in a method known as TLB shootdown [24]. TLB shootdowns are generated by a processor which issues an interprocessor interrupt (IPI) for remote call to TLB shootdown interrupt handler. The TLB shootdown interrupt handler then issues local TLB invalidation. On top of that, TLB shootdowns are synchronous and block further progress until all cores have completed the interrupt handling. TLB shootdowns are very expensive especially with a high number of cores and sockets. For example, LATR [89] measures an order of magnitude which ranges from 6 μs for 16 cores to 80μs for 120 cores.

Concerning the cost of TLB shootdowns in AutoNUMA, there are two main contributions. First, the background thread which changes protection of page table entries (pte_t) issues TLB shootdown to make protection changes visible on all cores. Second, during AutoNUMA page migration from one node to another, the TLB mapping information in all caches needs to be updated to the new mapping to prevent remote processors from accessing a stale version of the page. LATR [89] shows that TLB shootdown represents from 5.8 % with one 4 KB page, to 21.1 %, with 512 4 KB pages, of the overall page migration cost. The cost of TLB shootdown in AutoNUMA shows that automatic page placement does not come for free and may fail to improve performance if the shootdown overhead is not compensated by page locality.

3.5.4.3 Challenges on CPU-less NUMA nodes

New byte-addressable memory backends such as CXL attached memory, HBM or NVDIMMs can be managed as CPU-less NUMA nodes in the kernel. AutoNUMA provides a solution to automatic page placement for NUMA nodes comprised of both memory and CPUs, however these mechanisms have several design problems for CPU-less NUMA nodes. First, *task migration* can not be performed on CPU-less NUMA nodes since there are no cores to schedule the tasks on. Second, as reported in TPP [104], CPU-less NUMA nodes will never be assigned pages because no access will ever originate from this node. This prevents page demotion with AutoNUMA algorithm.

We discuss solutions for tiered memory NUMA nodes in the following paragraphs.

3.5.5 Automatic page placement of CPU-less NUMA nodes

As described in §3.5.4.3, AutoNUMA algorithm does not support demotion of pages to a CPU-less NUMA node. Automatic page placement on memory tiers need

to address the design problem in AutoNUMA page demotion algorithm for CPU-less NUMA nodes. Similarly to swapping or page cache mechanisms, automatic page placement needs to automatically place hot pages in local memory and cold pages on a tiered memory backend.

In *swap* and *page cache* page reclamation mechanisms, presented in section 2.4, pages are offloaded to a storage backend and need to be copied locally to be accessed or modified. On the contrary, in automatic page placement algorithms, processors can access or modify remote page content without copying pages locally. Thus, contrarily to page reclamation techniques, automatic page placement algorithms leverage byte addressability of memory tiers to avoid page IO and to deliver higher performances.

There exists different alternatives to automatic page placement on memory tiers. In this section, we focus on two different approaches to the problem of *selecting hot pages* in automatic placement algorithms. The first approach proposed by Meta is TPP [104] which leverages the existing LRU implementation of Linux for hot page selection. The second approach proposed by Intel named memory tiering [78] has been merged in the kernel and proposes a new algorithm for hot page selection.

3.5.5.1 LRU based memory tiering

TPP [104] proposes to rely on the LRU infrastructure for page demotion and to add cold LRU pages to a demotion list.

TPP motivates their contribution by identifying a bad performance pattern in AutoNUMA leading to page migration "ping-pong". Indeed, in AutoNUMA upon NUMA hint fault, a page is systematically promoted without further checks on its recent activity causing cold pages to get promoted before being demoted back shortly after. In order to solve this problem, TPP proposes to decouple allocation and reclamation watermarks by introducing two new watermarks allocation_watermark and demotion_watermark ($allocation\_watermark < demotion\_watermark$) on local nodes with CPUs **only**. In their new setting, background page reclamation begins when free pages is less than low_watermark. Page allocation is local when number of free pages grows back above allocation_watermark. Meanwhile, background reclamation carries on until number of free pages reaches demotion watermark. The extra *headroom* between allocation and demotion watermarks avoids pages coming back and forth. TPP achieves 18 % application performance improvement against default Linux policy and outperforms AutoNuma and AutoTiering by 10-17 % for web and cache workload. Interestingly, the authors report that AutoNUMA can even yield worse performance over default Linux policy.

3.5.5.2 MFU based memory tiering

Linux memory tiering identifies two drawbacks in the concurrent prototypes.

First, it identifies that TPP [104] approach based on the LRU to select hot pages can be improved. Indeed, the LRU algorithm is designed to identify cold pages but is not optimal to select hot pages [37].

Second, it identifies that the AutoNUMA approach which selects the most-recently-used (MRU) pages for promotion is not ideal either. In AutoNUMA MRU,

all accessed pages since the last scan period are recorded. But, AutoNUMA does not consider how many accesses have been made since last scan. Yet, scanning delays can be very long (up to 60s) leading pages rarely accessed to be observed accessed and thus promoted.

Linux memory tiering [78] proposes an approach based on the estimation of page accesses frequency as in most-frequently-used (MFU) policy. It uses the existing mechanisms of AutoNUMA for scanning and faulting presented in §3.5.3. Linux memory tiering implements the MFU policy by maintaining for each page a *hint fault latency* which represents the elapsed time between scan time and fault time. It is not strictly a MFU policy as it does not count how many accesses have been performed on a page. Instead, under the assumption that scan phases are repeated periodically, the hint fault latency captures access recency on each page. Accesses recency can be considered as a good approximation of access counting as hot pages are more likely to have shorter page fault latency.

There exists other methods in the literature to identify the set of accesses or dirtied page tables. For instance, Hemem [122] has proposed to use processor hardware sampling (Intel PEBS) for page tracking. However, a common issue with the other prototypes is the large overhead introduced by the tracking method. On the contrary, the methods which have been presented in this section introduce decent overheads and rely on software implementations which makes them usable for virtualization [131].

In-kernel automatic page placement adds a considerable advantage over explicit memory management by offering a transparent solution for software developers. However, the techniques achieve application-specific and limited performance gains. In the next section, we briefly present the reasons for these limits and user-space alternatives.

3.6 In-user automatic memory placement

In previous section, we have reviewed automatic memory placement techniques available at kernel level through instrumentation of page faults. However, with language runtimes there exists other opportunities to retrieve access information on objects and migrate them transparently using compaction or garbage collection phases. One of the advantage of in-user automatic object placement is to work directly at object granularity instead of page size which is usually too large for placement. Thus, in this section we detail this language granularity semantic gap before a short overview of user-space automatic object placement.

3.6.1 Object granularity and the limits to page granularity

There exist a significant gap between kernel knowledge of memory accesses and the knowledge of memory from language runtimes or applications. In general, language runtime know a lot more information about memory than the kernel which makes language runtime a more appropriate layer to implement some of the automatic object placement strategy than the kernel.

Typically, programming languages directly handle objects of any size and usually significantly smaller than page sizes. This enables finer-grained IO and prevent memory copy amplifications already identified in swap techniques by AIFM [125]. Panthera [154] further observed that relying exclusively on physical pages accesses tracking may lose object access information because multiple objects can be colocated on a same page. Programming languages also maintain object relations between each other under the form of a reachability graph where in-use objects can be found from a limited set of root objects. All this information can be leveraged by languages to propose smarter object placement compared to kernel placement.

In the next paragraphs we discuss the proposals for automatic object placement in user-space.

3.6.2 Object placement in user-space

Object placement prototypes for heterogeneous memory There exists different solutions for object placement on heterogeneous memory. Some of the solutions rely on language runtime information and services to implement placement policies while other solutions are more intrusive and require explicit source code modifications.

3.6.2.1 Language runtime object placement

Additionally to providing more knowledge, managed languages also offer additional memory services such as compaction and garbage collection which can be extended to include placement decisions on heterogeneous backends [6, 154].

For instance, Shoaib et al. [6] is one of the first prototype to propose object migration based on their access. In their work, they propose to maintain in main memory most frequently accessed objects and to move in a NVDIMM memory tier objects less frequently written to.

Panthera [154] proposes automatic object placement for big data framework workloads (e.g. Spark) in a server using DRAM and NVDIMMs. Their work relies on the observation that the access pattern and lifetime of objects can be observed easily in big data frameworks. They contribute two mechanisms: First they propose a static analysis tool for inference of data access pattern on Spark persisted data-structures (persisted RDD) to determine if objects should be allocated or moved to DRAM or NVDIMMs. Second, they propose to extend the Parallel Scavenge garbage collector in openJDK to implement promotion and demotion of objects from DRAM to NVDIMMs based on access pattern inference.

3.6.2.2 Explicit object placement

Based on the observation that page granularity leads to heavy IO amplification, AIFM [125] introduces new user-space abstractions to perform remote memory accesses. They propose *unique remotable pointers* as an abstraction for single pointer references to the object, and *shared remotable pointers* for when multiple references to an object are held. It is interesting to review AIFM because their approach to remote memory accesses in user-space is inspired by Linux swapping but enables better performances.

The remotable pointer abstraction requires source code modifications but it claims to require limited changes to application logics to support remote memory accesses. Indeed, the abstraction is based on the semantic of C++ pointers which are widely used in multiple applications. Additionally, the usability of their abstraction has been demonstrated by another research paper [174] which has implemented fault-tolerance for remote memory based on AIFM remotable pointers.

Remote pointers manage the state of each object by encoding metadata information directly in virtual addresses (on unused bits 47 to 63). It reproduces many of the information found in a x86 page (pte_t) with a *dirty* byte [2] to track object mutations, a *present* bit to track the presence of the object in local memory and a *hot* byte updated using GC-like barriers [76]. The hot byte is similar to the access bit in x86 pages. Additionally to similarity with hardware page bits, remotable pointers also provide information found in kernel pages (struct page). In details, remotable pointers encode a *shared* bit to indicate that the reference counter of the pointer is higher than 1 and an *evacuating* bit to signal that the object is currently being evicted.

Upon memory pressure, AIFM evicts a set of objects identified as less frequently accessed. The set of cold objects is identified by running a replacement algorithm which scans hot bits contained in remotable pointers similarly to Linux LRU. AIFM implements a replacement algorithm based on CLOCK algorithm [35], which has many similarities with Linux LRU, to identify the set of cold objects. AIFM implements object evacuation by using TCP sockets to send them on a remote memory server.

One of the challenges in the use of AIFM remotable pointer is to find the proper timing to issue the evacuation. Indeed, in AIFM a remotable object which needs to be evicted may still be referenced by multiple other objects and there exists no

[2] It is not an error, it is a byte

backward reference to track parent objects. In Linux swapping, this is easily solved by clearing the present bit of the page (pte_t) when it is evicted so that any future access causes a page fault generated by the MMU. However, AIFM does not modify page table management and can not rely on this technique to handle accesses to an evicted object. Instead, AIFM forces the use of a Dereference Scope for every dereferencement of a remotable pointer. Dereference scopes are objects allocated on the stack which call a destructor method when the stack frame is destroyed. The idea is borrowed from C++ Resource Acquisition Is Initialization (RAII) used for resource release and automatic unlocking of mutexes. In AIFM, RAII is used to defer the eviction to the termination of the scope of a remotable object.

One of the remaining problem in AIFM is to offer pauseless eviction of objects. Pauseless eviction is inspired by pauseless garbage collection [76]. Pauseless eviction means supporting safe concurrent evictions of objects from the runtime evacuator threads to object mutations performed by mutator threads. Similarly to concurrent GC, AIFM uses a barrier based on object evacuation bit to synchronize mutators and the evacuator. The barrier manages concurrency on the evacuation bit using RCU synchronization [136] (think of it as a read-write semaphore). RCU is known to cost little overhead for the RCU writer and to favour RCU readers.

The user-space memory management approach used in AIFM prevents IO amplification and page fault overheads. This enables AIFM to deliver 61 times higher performances than fastswap, a remote memory prototype integrated with Linux swap system.

In this section, we have reviewed the different mechanisms available to report heterogeneity for the operating systems. We have seen that topology reporting information are focusing on a reduced set of properties with average latency and bandwidth with no consideration for the differences between read and writes.

After reviewing topology exposition, we have reviewed the different solutions offered by the kernel to manage heterogeneity. In particular, we have presented MGLRU, the updated LRU algorithm in Linux to handle multiple memory tiers. We have also presented remote caching of memory on remote servers which proposes to offload specific subset of memory (page cache or JIT compiled code) on a remote server to minimize the performance cost. Then, we have discussed explicit memory management to let applications directly manage a memory tier deprived of kernel services. We have discussed how explicit memory management could hurt application memory management transparency. Next, we have discussed existing approaches to maintain legacy memory abstractions on a tiered-memory system. We have presented some of the existing in-kernel prototypes which try to perform automatic page placement on heterogeneous systems. Our review of these prototype separates solutions for classical NUMA nodes comprised of CPUs from NUMA nodes acting as tiered memory with no compute capability. Finally, we have discussed some of the advantages of in-user memory placement since these prototypes directly provide significant performance gain by leveraging additional information unavailable at kernel level.

This section has presented different meaningful contributions to memory management on heterogeneous systems by leveraging some of the description from chapter 2. In the next section, we present contributions to memory disaggregation not limited

to memory management. In particular, we will focus on key design challenges of the different memory disaggregation prototypes with a review of hardware-level, OS-level and database solutions.

4 Architectures of disaggregated memory systems

Previous section has reviewed the literature on memory management for heterogeneous backend. In this section, we propose to review main architectures and techniques used in the literature to implement disaggregation of memory resources.

Our review first begins by presenting how it is still possible to perform remote memory accesses since cache-coherent interconnects are not yet available on the market. Thus, we present RDMA, a common networking fabric which has been popularized in datacenters for its speed and low CPU overhead. Second, we discuss the limits to RDMA usage in datacenters with scalability and performance considerations. Third, we discuss how OS-level prototypes may leverage different mechanisms to offer transparent remote memory accesses. Fourth, we review distributed shared memory (DSM) prototypes which try to support remote memory sharing across multiple nodes, a useful service which is not always considered in memory disaggregation. Finally, we discuss some of the hardware solutions proposed to speed-up remote memory accesses for applications.

4.1 RDMA: A fabric for memory disaggregation

Remote Direct Memory Access (RDMA) is a set of protocols which targets rack-scale or datacenter scale networking communications. Most common RDMA protocols include Infiniband, iWarp, ROCEv2, Tofu. Infiniband and ROCEv2 remains the most widely used and commercially available protocols.

All RDMA protocols are based on the idea of offering a networking primitive which uses the remote node DMA controller instead of its CPU as in classical NIC. RDMA initially targeted High Performance Computing workload. It has been made easy to program thanks to middleware which have implemented high-level abstractions on top of it (e.g. Message Passing Interface [105]).

RDMA performance gains over the widely used Ethernet Network Interface Controller (NIC) are achieved by various factors. First, physical, link, networking and transport layer are implemented in the hardware of the RNIC which enables faster processing thanks for networking stack operations than CPU networking stack operations. Second, RDMA exposes a userspace API for data-path operations which prevents the expensive privilege level changes required to perform in-kernel network-

ing. Third, RDMA greatly benefits of its one-sided communication mode where data may be transferred between two nodes without notifying the remote CPU. This prevents expensive interrupt handling to be performed during communications, although it is sometimes inconvenient not to be notified of the reception of a message. Fourth, RDMA offers an asynchronous programming model which enable good communication overlaps. Fifth, RDMA API enables easy zero-copy communications which is a key advantage for large size messages but less useful for smaller size buffers.

4.1.1 High-speed hardware networking stack

Infiniband protocol is composed of multiple layers of encapsulation namely a link layer, a network layer and a transport layer. The link layer implements a *flow control* service. Infiniband transport layer offers different service modes with different guarantees such as *reliability* which guarantees that messages are acknowledged and *connected mode* which only supports one-to-one communications between the two connected queue pairs (contrarily to Datagram for one-to-many communications). Available infiniband transports include Reliable Connected (RC), Unreliable Connected (UC), Unreliable Datagram (UD), Reliable Datagram (RD).

4.1.2 User-space networking

One of the main benefits of RDMA networking stacks is to rely on userspace libraries for most networking primitives. Part of the control path such as registration of memory region still requires going through the kernel, but message posting is fully issued from userspace. This is an important performance gain given the overhead associated with privilege level changes in the kernel emphasized by mitigations patches for Spectre-like vulnerabilities [65].

4.1.3 Freeing Remote CPU cycles

Similarly to a DMA controller, RDMA network cards offer an operation mode named *one-sided*, which does not involve target CPU usage for message reception. This mode yields great benefits to memory mutualization as it does not use the target's CPUs which can be used to run other jobs. As a consequence, one-sided operations do not generate an interrupt on the target CPU after a message has been written in target RAM.

4.1.4 Asynchronous data-path API

RDMA offers an asynchronous interface to the developer. This means that the developer can post new requests without waiting for the termination of the previous one.

RDMA asynchrony relies on the *queue pair* (QP) abstraction for communication. In a way, queue pairs try to offer a similar abstraction for communication as BSD sockets by maintaining queues for communications, context for messages and maintaining a notion of session between peers. Queue Pairs are made of a *submission queue* (SQ) to post requests and a *completion queue* (CQ) for notification of the

completion of the request. In kernel RDMA stack, completion queue entries may be retrieved through polling-based notifications (CPU expensive) or interrupt-based notifications (latency expensive).

4.1.5 Zero-copy IO

RDMA also provides the ability to use any buffer of data to forge the message payload without copying it. This is an important difference with traditional kernel TCP stacks where user-space buffers are copied in a DMA-able zone of the kernel which is used to forge the payload of the message. Zero-copy communications save a memory copy of the buffer but may require mutual exclusion between NIC and CPUs for concurrent accesses to the buffer.

4.2 Limits to the adoption of RDMA in datacenters

We have reviewed the key advantages of RDMA network cards for communications over more conventional protocols such as TCP and UDP built on IP networks. However, despite its advantages RDMA remains underused in datacenter communications. In this section, we discuss the existing limits to the adoption of RDMA. First, we present scalability issues posed by RDMA networking for datacenter networking. Then, we discuss some of the practical limits in terms of security or usability of RDMA. Finally, we discuss current communication limits which are expected to disappear with new hardware interconnects.

4.2.1 Scalability issues

One of the reason RDMA is not widely used in datacenters is due to performance degradation (bandwidth and latency) when the number of nodes used in the communication grows. The scalability challenges of RDMA mostly appear when the metadata managed by the RDMA card becomes heavily bigger than the internal hardware caches of the card. Various works [145] have reported multiple scalability issue in Infiniband NICs. We present scalability problems caused by the increasing number of queue pairs and memory managed by the NIC.

4.2.1.1 Queue Pair scalability

One of the identified scalability issue is the increase of RDMA operation *latency* with the number of queue pairs created in the NIC. Various works [145, 46] have identified that RNICs performance does not scale with the number of queue pairs. This is caused by additional metadata maintained for the management of queue pairs which may become larger that RNIC cache size. This phenomenon is mostly reported for RDMA Reliable Connected mode.

Lite [145] proposes *Queue Pair Sharing* at software level to offer better scaling performance.

4.2.1.2 Memory Tables scalability

One of the problem of RDMA NIC is that it operates on physical address space (or IO virtual address space) but applications manipulates virtual addresses. In order to solve this problem, RNIC vendors have shipped a translation unit in RNICs named *Memory Translation Table* (MTT). MTT is structured as a hashmap which associates virtual addresses with IO virtual addresses (iova). Similarly to the MTT, RNICs rely on a Memory Protection Table (MPT) to enforce access control on RDMA memory accesses. RDMA proposes a *memory region* abstraction, which implement provided both *lightweight access control* service for memory accesses (MPT) and *address translation* to push address space mappings to the RNIC (MTT).

Lite [145] observes that MTT and MPT may not fit in RNIC cache when the number of mapping grows. Indeed, each RNIC cache miss causes a DMA read from main memory to bring required MTT entries in the RNIC cache. RNIC cache miss causes RDMA write latency to be degraded from an average of 2 µs for less than 100 memory regions to almost 4 µs for $100,000$ memory regions.

Additionally to the number of memory regions, Lite [145] observes that the size of memory regions also causes slowdown because of MTT cache trashing. They observe that for memory regions above 4 MiB, throughput is lowered and can be divided by two for memory regions above 64 MiB.

4.2.2 Other practical limits

There are other reasons for the low adoption of RDMA in the datacenter. For instance, RDMA provide isolation between physically contiguous memory buffers by using a hardware check on a 32-bit security key. This security solution is efficient but provide weak isolation since an attacker can still try to guess security keys to access memory of another host.

There exists other practical limits such as the fact that the API remains complex and require the developer to implement multiple data-structures for communications which are usually hidden in kernel drivers (e.g. ring buffers).

4.2.3 Unnecessary round-time-trip

DirectCXL [60] observes that RDMA READ operation requires at PCIe level two Memory Read (MRd) Transaction Layer Packets. One transaction is used to fetch a message descriptor, the second message is used to read data. On the contrary, interconnects offering memory mapped interface like CXL can perform a read in a single DMA READ operation.

4.3 Mechanisms for OS-level transparent remote memory accesses

In previous section, we have discussed how RDMA can be used for remote memory accesses and its limits which justify the needs for new rack-scale interconnects.

Meanwhile, prototypes may rely on RDMA however they still need to offer transparent remote memory accesses (with no application modifications) at operating system level. A simple, yet, widely used solution is to perform IO operations in page fault handling during translation of virtual addresses. In this section, we present some of the mechanisms available to implement remote memory accesses at OS-level with. Then, we review the identified limits to these mechanisms.

4.3.1 Swap based accesses

Various works [57, 7, 61] rely on Linux kernel swap for remote memory access. Indeed, swap offers a natural extension to the limited physical address space while maintaining transparent virtual memory accesses. It is a convenient interface because, as presented in section 2.5, it naturally relies on Linux LRU service which distinguish hot pages from cold pages. Additionally to page hotness identification, Linux swap also integrates the mechanism which forges the IO request containing the set of cold pages to be written on the swap backend. In the following paragraph, we present block device and frontswap swap backends.

4.3.1.1 Block device

Block devices are the historical interface to swap system in Linux. Block device are the logical abstraction used to represent the sector layer of block storage devices. Thus, various remote memory works such as Infiniswap and Gao et al. [57, 61] rely on this interface to implement remote memory accesses. In these works, a memory server is running to perform a large buffer allocation. A kernel module serves a memory client by implementing a block device interface. The block device receives an IO requests to read or write a 4 kiB page on the remote backend and translates the IO requests to whatever communication mechanism is used. For instance, Infiniswap [61], translates IO requests to RDMA requests to propose a production-ready solution for transparent accesses to rack memory.

Instead of proposing a production solution, Gao et al. [57] rather propose a simulation prototype by injecting configurable latency and throughput in the block device. Then, they observe the consequences of the performance degradation on various applications to draw application performance profiles depending on how much remote memory is used in the application.

4.3.1.2 Frontswap

Fastswap [7] uses another swap interface named frontswap. Frontswap is an interface integrated in the swap path of page fault handling. A frontswap backend acts as a cache for an underlying device backend. Contrarily to block devices, the frontswap abstraction does not directly manage the available size of the swap backend. Indeed, frontswap acts as a cache layer, thus it swap sector allocation is performed on the underlying backend registered below frontswap in the kernel stack. Thus, frontswap is not aware of a swap size. Moreover, frontswap offers a **blocking** API for **single page** operations only. The frontswap API supports load, store, and invalidate calls.

4.3.2 Swap based performances

Gao et al. [57] proposes a block device backed by local DRAM and inject artificial delays and evaluate application performance degradation. They show that for most applications using 40% of local memory still provide acceptable performances. They give order of magnitudes around 3 to 5 μs latency and 40 Gbps bandwidth for reasonable performance degradation.

4.3.3 userfaultd, page fault handling in user-space

Previous section has reviewed the swap interfaces available for transparent virtual memory accesses. Another solution available is userfaultd, a kernel mechanism which allows a userspace process to register for notifications of page fault handling. Userfaultd offers a more generic approach than swap since it offers the resolution of the entire page fault to the process. One of its advantage over Linux swap is the possibility to manage the anonymous memory of a single process as a cache instead of the anonymous memory of the entire system.

Fluidmem [29] relies on userfaultd a mechanism, which brings page fault management to the userspace. The userfaultd userspace callback mechanisms allows a process to register a function to be called by the kernel during page fault handling. The feature was initially designed to help implement virtual machine post-migration logic by letting a userspace hypervisor (e.g. qemu) located on a destination node fetch pages from the source node.

Fluidmem [29] achieves full memory disaggregation by modifying qemu VMM to trap guest memory page faults transparently for the VM. It is able to save a few CPU cycles in page fault handling by avoiding to go through the various cases of the kernel swap system. However, the page fault handling logics in user-space remains mostly similar to swap-based techniques and thus suffers similar limitations.

4.3.4 Limitations to swap based disaggregation

Multiple prototypes for disaggregated memory have been proposed over the last years based on page fault instrumentation. Consequently, many papers have shed lights on the biases introduced by this technique which is now considered a poor candidate for remote memory accesses even for simulation purposes.

4.3.4.1 IO amplification

AIFM [125] shows that page size granularity cost is not observed in the transmission time but in the *IO amplification* phenomenon where objects of smaller size are collocated on the same page.

4.3.4.2 Analysis of swap cost in Page Fault

AIFM [125] also provides an in-depth analysis of the cost of swapping-in a page from a SSD backend. They observe that despite an incompressible 6 µs hardware read latency for 4 kiB pages, swapping in a page costs an average of 15 µs which

means 11 µs of overhead introduced by the swap mechanism. AIFM proposes a time analysis of the different time-contributions due to swapping in a page. They show that performing the IO (6 µs) and spinning for completion costs up to $11.7s$ which is the **main contribution** and shows that kernel IO stack is already responsible for 5.7 µs of overhead. Furthermore, they show that trapping to kernel costs less than 1 µs while the allocation of the page table entry contributes to around 1 µs. Finally, the creation of the mapping in the page table costs around $2s$.

4.3.4.3 Anonymous memory only

Registration of a swap device can only serve to back anonymous memory pages while cooperative caching techniques (see §3.4.1) have shown that potential benefits may be found in disaggregating the page cache as well.

4.3.4.4 No parallelism

One of the problem of swapping we have identified without finding alternative sources is that in background reclamation context, pages eviction are performed sequentially unless multiple NUMA nodes are used. Only direct reclamation (memory reclamation in the context of a page fault, see section 2.4) offers *parallel eviction* and *parallel reads*.

This is problematic when swapping becomes a common memory management technique as it is used in disaggregated memory relying on it.

4.3.4.5 No sharing

Swapping relies on dedicated disk partitions, and it requires exclusive access to the backend. This means that, in the case of memory disaggregation, a swap backend can not serve as a solution for shared memory between multiple compute nodes.

4.3.4.6 TLB shootdown

Linux kernel swap mechanism relies heavily on TLB shootdowns (see §3.5.4.2) Indeed, after swapping out a page, a TLB shootdown is sent to all cores having the mapping in their TLB to guarantee the visibility of the cleared present bit and the updated pte_t which encapsulates sector location information on the swap backend. ecoTLB [103] has measured up to 18% overhead due to TLB shootdown in memcached using infiniswap [61]. This is a major design argument against swap-like techniques compared to methods which rely on hardware caches.

4.3.4.7 The hidden hardware cost of page faults

As explained in Kona [28], there exists an intrinsic cost to handle page faults which makes them irrelevant for latency-sensitive workload. Indeed, when a page fault is delivered, the processor requires to flush its instruction pipeline. Then, the page fault handler code performs various memory accesses and pollute CPU caches.

Finally, Kona has identified page fault as implicit barriers for the CPU prefetcher as no prefetching can be performed past the page fault.

This section has presented common mechanisms used to support transparent remote memory accesses at operating system level. However, all the solutions reviewed do not propose support sharing of remote memory between multiple nodes. The next section presents some of the contribution to implement cache coherency protocols at OS-level and application level to support remote memory sharing.

4.4 Distributed Shared Memory

The previous section has presented various mechanisms which can be used to implement OS-level transparent remote memory accesses by relying page fault instrumentation. This approach has also been extensively used in the 1990s to implement in-kernel cache coherency protocols to support sharing of memory across multiple servers.

Distributed Shared Memory is an architecture which leverages compute resources from different node to issue transparent memory accessed on a uniform address space. In DSM, the uniform address space spans on memory from multiple nodes. Prototypes presented in the following section (section 4.4) propose variations in the design of cache coherency protocols.

In this section, we first review page-based DSMs which implement a distributed shared ***virtual*** *memory by modifying page fault handling mechanisms. Second, we review the use of object-based DSM commonly used in databases to support distributed transactions across a set of multiple nodes. Third, we discuss how RDMA can accelerate DSM implementation before reviewing hardware acceleration for distributed shared memory.*

4.4.1 Page-based DSM

Before the arrival of NUMA processors, various software proposals have tried to transparently scale the shared memory abstraction to multiple nodes. These prototypes propose rely on modifications to page fault handling to support coherent memory accesses.

4.4.1.1 Shared virtual memory

Some DSM approaches [96, 30, 170] transparently expose a virtual memory to maintain application transparency.

Ivy [96] first proposed an algorithm to implement a coherent shared virtual memory by relying on page fault handling. They propose an algorithm to let an OS process virtual memory span over each node physical memory. The use of pages is based on the observation that sending messages larger than object size does not significantly increases additional in transmission cost. It is important to note that this assumption is already arguable for pure message exchanges, but, information tracking and page thrashing has been shown to be expensive at page size in later works (see §4.3.4).

GiantVM and aggregateVM [170, 30] leverages hypervisor page fault handling to expose a shared virtual RAM resource for a virtual machine. They expose a shared guest physical address space implementing Ivy's algorithm in EPT fault handing in KVM.

4.4.1.2 Page-based cache coherency

Ivy [96] discusses possible strategies to solve memory coherency problems with a classification in two dimensions. First dimension describes two approaches to *page*

synchronization: writeback and invalidate. In the **writeback approach**, when a core performs a write on a page, it is propagated through a transaction to all cores having a copy of the page to update it. In the **invalidate approach**, when a core performs a write, it acquires page ownership and invalidates all the other copies. Then, it performs the modification on the page block.

Second dimension describes different approaches for *page ownership*. It presents **static ownership** where a processor owns a page permanently as opposed to **dynamic ownership** where page owner changes over time.

A possible dynamic page ownership strategy is to rely on a centralized monitor where a single monitor is responsible for managing a table of pages with their associated lock to serialize conflicting requests and owners of pages. However, this static ownership approach is bottlenecked by communications towards a unique monitor.

Thus, another strategy is to implement dynamic page ownership with distributed monitors. The idea is to distribute memory pages across monitors, each monitors having the responsibility of tracking ownership for its set of pages. Naive distributed approaches require more messages to locate the owner than in the centralized approach, however Li et al. noted potential gains possible by replicating copyset foretelling directory-based cache coherency.

4.4.1.3 Limits to page-based DSMs

Software DSM often struggle against hardware implementation because few prototypes propose alternative memory models to sequential consistency. For instance, Ivy [96] and giantVM [170] offer *sequential consistency* while modern commodity processors propose weaker memory models. There exists proposals (usually in database systems) to integrate weaker models, such as GAM [27], which proposes partial store order and shows that it can deliver a higher number of memory operations thanks to operation pipelining.

Yet another problem in the design space of page-based DSM can explain their little usage in production. The first problem in page-based DSM shown by Kona [28] is the use of page fault handler to implement cache coherency protocol. Indeed, as described in§4.3.4.7, page fault handling is an expensive interface between hardware and software, which breaks the flow of execution and prevent the use of many hardware mechanisms (prefetching, instruction pipelining, cache hit).

The second problem shown by Kona [28] is that few memory accesses are performed at page size, which has consequences in identifying dirtied memory for writeback. They estimate that tracking dirtied pages instead of dirtied cache lines results in 2 to 31 more data to be written back.

Based on the problems introduced by page granularity and page fault handling, we propose to review in the next section other software DSM approaches based on object granularity.

4.4.2 Object-based DSM

The concept of distributed shared memory has a long research history in the database community. Indeed, in database architectures, there is a long ongoing

discussion around distributed shared-nothing architectures and distributed shared-storage or distributed shared-memory architectures. [138] In the 1980s, distributed shared nothing (DSN) has taken over DSM architectures but recent work [27, 157] are making a case to reboot distributed shared memory architectures again based on the advances in interconnect speeds. In next section, we review GAM database, the state of the art distributed shared memory implementation, which offers object granularity memory accesses. Then, we discuss the limits to current software DSM systems which have led to the adoption of hardware acceleration.

4.4.2.1 GAM, an efficient DSM database using RDMA and caching

GAM [27] is an interesting DSM database prototype because it implements many efficient mechanisms missing in page-based DSM prototypes.

First, GAM imitates the architecture of page-based DSM by relying on locally managed software caches for a shared pool of memory. GAM manages memory at the granularity of cache lines and implement a dedicated LRU cache line replacement algorithm. The software cache layer enables shorter memory accesses.

Second, GAM relies on directory-based cache coherency protocol instead of snooping cache coherency to support scalability to multiple nodes. GAM directory-based cache coherency protocol uses a global coordinator, which communicates with the other agents using RDMA.

Third, GAM offers a weak consistency memory model named *partial store order* [64] to support write reordering and achieve a higher operation throughput.

There exists concurrent prototypes similar to GAM [46], although this section mostly proposes a short overview of a larger set of contributions in databases since 2015. There have been multiple proposals to support distributed transactions using RDMA to implement DSM at object granularity. The different contributions to transaction processing proposed by these systems are out of the scope of this thesis.

4.4.2.2 Superlinear cost of cache coherency

Software object databases yield significant performance gains compare to their page-based counterparts. However, databases typically require to store more and more data which require to use resources from more and more nodes. Thus, a remaining question in object-based DSM databases concerns the performance impacts to scale to larger pool of resources.

In Concordia [156], the authors try to evaluate how cache coherency scales with the size of shared memory. They study the state-of-the-art DSM database named GAM [27] (see §4.4.2.1). GAM is a fair choice for testing the scalability since it uses multiple advanced technique known to offer best possible scalability such as RDMA for fast-communications and *directory-based cache coherency* (see §1.5.5.3). Despite all GAM advantages, Concordia shows that application throughput **degrades superlinearly** with the size of shared data. Concordia explains this superlinear degradation by the number of cache coherency messages between nodes.

The limited adoption of software DSM systems is caused by various factors. Some limits can be circumvented with more advanced software solutions by providing alter-

native memory consistency models to sequential consistency. However, there exists also fundamental limits software DSM. For instance, a transparent software DSM causes undesirable amplification of IO messages due to granularity mismatch between objects and pages. Moreover, software DSM implementations offer limited scalability because they require additional round-trips which can be prevented at hardware level (see details in section 4.5).

4.5 Hardware accelerators for disaggregated memory

In the previous section, we have presented various prototypes which supports a distributed shared memory abstraction. We have shown how DSM abstraction enabled multiple execution units with a caching layer access a shared pool of memory coherently. However, all of these prototypes even when they use high-speed fabrics such as RDMA suffer unaffordable performance degradation. It is possible to obtain significant speed-ups by moving some of the requirement of DSM to the hardware. A production example of the memory hardware offload is illustrated in NUMA machines which are now broadly used and which implement hardware cache coherency protocols. In this section, we review how remote memory accesses may benefit from offloading memory services to the hardware. We first look at DSM acceleration with hardware cache coherency before reviewing hardware acceleration for non-coherent remote memory accesses.

4.5.1 DSM acceleration using in-network cache coherency

Concordia [156] proposes to accelerate DSM using network switch. They propose *FlowCC* a Write-Invalidate protocol with in-switch acceleration which requires a single round time trip for coherency. On a programmable network switch, Concordia implements two primitives: *multicast invalidation* and *Lock-Check Forward (LCF)* pipeline to enforce in-switch coherency. The switch stores an array indexed by cache blocks containing a lock, a global status and the copyset (nodes holding the block). First, Lock-Check-Forward pipeline serializes concurrent cache block operations issued by cache agents using a 16-bit read write lock. Read lock is acquired on a read miss and write lock is acquired on a write miss to a DSM entry. Second, the pipeline filter out invalid requests which occur for concurrent requests to a same cache block. Finally, the switch performs request forwarding to cache agents for invalidation or to home agent for reading from global memory. Concordia achieves performance speed-up over a software DSM from 2 (graph engine) to 4 (KV-store) times.

4.5.2 non-coherent disaggregated memory using snooping FPGA

Kona [28] also reports unacceptable latencies of using transparent virtual memory techniques to perform remote memory accesses. Kona identifies *page fault handling* and *IO amplification* as key limitations to perform remote memory accesses in the kernel. Thus, it proposes to implement a writeback cache with host memory for a large pool of remote memory. They identify three different problems to implement the writeback cache and propose solutions for them.

First, Kona requires a mechanism to **fetch remote cache lines** in the cache. Kona uses a fake memory device built in the FPGA and attached to the physical address space of the host. Kona does not issue page faults when it tries to read remote memory, instead the FPGA receives a cache miss requests and directly reads

remote memory. This approach to fetch remote memory avoids TLB shootdown since MMU page table remains unchanged.

Second, Kona requires a mechanism to **track dirty cache lines** to determine which cache lines need to be written back instead of writing back the full 4 kiB page. Kona proposes to leverage the snooping FPGA to report dirtied cache lines instead of using expensive write-protection of pages.

Third, they need a mechanism to **evict local cache lines** to remote memory. Kona only needs to evict cache lines which are dirty. Dirty cache lines are aggregated in a dirty log and evicted using asynchronous one-sided RDMA (from the FPGA).

All these optimization enable Kona to achieve 1.7 to 5 times shorter accesses and to reduce dirty data amplification by 2 to 10 times compared to page-based approach like infiniswap [61].

This section has provided a very short overview on how hardware may help speed-up existing software approaches. The first approach provides cache-coherency implementation in a switch to speed-up software cache coherency in DSM systems. The second approach relies on hardware to support remote memory accesses and dirty memory tracking at fine-grained granularity to prevent misidentification of dirty data and IO amplification.

In this section, we have reviewed architectures and techniques used to implement disaggregated memory. We have started this discussion with a focus on how to achieve remote memory accesses with RDMA, a fabric for remote memory access which is gaining more and more adoption in datacenters. We have seen that RDMA has served as an early solution for page-based disaggregated memory prototypes and which is still relevant to implement object-based software distributed shared memory. We have also seen existing limits in scalability with RDMA which is major challenge for datacenters. Moreover, there exists design limits with RDMA such as the lack of cache semantics (CPU load/store) and additional round-trips for read operations which have been solved in new interconnects such as CXL.

Then, we have reviewed the interface used by most OS-level prototypes for disaggregated memory to offer accesses to remote memory through instrumentation of virtual addresses. We propose a review of software DSM prototypes which consider disaggregated memory as supporting sharing of remote memory across multiple servers. In particular, we have shown that page-based DSM offered limited performances and were rarely used on the contrary of software object-based DSM which are being used in databases.

Last we have seen how hardware can help to provide useful services for remote accesses with hardware cache coherency to speed-up software DSM or dirty tracking to generally reduce IO amplification in systems leveraging a writeback cache for remote accesses.

This section concludes our review of the heterogeneous memory landscape before the beginning of the last section which present the advances in VMs usage in datacenter to reduce resource consumption.

5

Virtual Machines Resource Management in datacenters

In the previous sections, we have reviewed memory management challenges in heterogeneous systems and presented the different solutions proposed. The study of virtual machines components and their use in datacenters has been left as a separate section since it introduces an additional level of complexity with system-level memory management considerations and rack-scale memory management challenges.

Initially, system virtual machines were proposed in the seventies [118] (e.g. cp-cms or IBM system370) to permit a single host to recreate a hardware execution environment with its IO, memory and compute resources. In the late 1990s, the emergence of NUMA architectures has restarted the interest in virtual machines. Indeed, operating systems did not scale well on multiple processors, Disco [26] proposed to run independent OSes on each NUMA nodes A few years later, Xen [19] proposed a VMM to run a hundred VM on a single server with no hardware virtualization support at the time. The next years have seen the emergence of datacenters with virtual machines becoming a standard abstraction for deployments. Datacenters have been able to increase resource mutualization while keeping strong resource isolation. Over the years, virtual machines have also received the help of processor hardware accelerators which are efficient enough to become a large scale deployment unit in datacenters. Newer and lighter abstractions like containers have emerged since then with cloud providers massively shifting to selling a large set of services instead of machines. Yet, virtual machines remain interesting when strong isolation is required or when migration of software stacks is desirable.

In this section, we first propose to review how virtual machines offer an execution and isolation unit for the datacenter with a short introduction to the implementation of modern virtual machines. Second, we discuss how heterogeneous memory can be transparently used in virtual machines. Third, we discuss challenges and solutions to achieve collaborations of guest and hypervisor memory management. Fourth, we present a short model to understand orchestration of resources at datacenter level with problems related to resource usage. We use these models for our discussions in the following sections. Fifth, we discuss the need for dynamic resource usage in VMs and solutions currently used. Sixth, we present transient virtual machines, a class of VMs which tries to improve resource usage by using resource leftovers. Finally,

we discuss orchestration and design of VMs using remote memory in the rack.

5.1 Virtual Machines: execution and isolation unit

Datacenters rely on the possibility to share hardware resources between multiple customers to propose affordable execution environments to customers. VMs propose a nice abstraction for cloud provider with interfaces for migration between hypervisors, checkpointing or resource allocation.

First, we review core services offered by VMs which explain their adoption in datacenter as isolation and deployment units. Second, we review how virtualization offers isolation of the execution of instruction blocks. Third, we present memory management mechanisms in VMs using software and hardware solutions. Fourth, we present how VMs expose IO devices and the challenges to manage them.

5.1.1 Core services offered by VMs

In the following paragraphs we review some use cases covered by VMs. In particular, we discuss possibilities offered for security isolation, for deployment and hardware emulation.

5.1.1.1 Security isolation through VMs

Operating systems rely on *process* as the main execution unit of user-defined tasks. Processes enable users isolation on the same machine or to separate tasks of an identical user with different security level.

Processes by design provide basic security guarantees with independent virtual memory address spaces to minimize the attack surface offered by memory sharing and avoid malicious processes from hijacking other processes.

However, processes remain vulnerable to *denial of service* from other malicious processes which may want to overuse computation time or memory capacities.

5.1.1.2 VM as deployment units

Available hardware capacities in server is growing at a quick pace. Software stacks are also less monolithic and managing compatibility of multiple version is sometimes hard. The industry has widely adopted the principle of shipping entire software stacks in packed deployment units. Although, this packed deployment units heavily rely on containers, it sometimes also necessary to ship specific operating systems with the deployment. Thus, some approaches use lightweight virtual machines for this task [149, 3] with the additional benefit of shipping the appropriate kernel with the deployment unit. For example, in computer networking, internet service provider are building emerging infrastructure services by hosting network services (firewall, packet inspection, ...) in virtual machines. It is known as Virtual Network Function (VNF).

5.1.1.3 Emulation of hardware interfaces

Virtual machines aim at providing similar execution environment as in a bare-metal setting. Virtual machines emulate hardware interfaces to support the execution of any unmodified operating system.

The software permitting the execution of a virtual machine is commonly named *virtual machine monitor* (VMM). Some widely used modern VMM are qemu [121] or firecracker [3].

Hypervisors rely on hardware accelerator components to tackle some bottlenecks of hardware virtualization. Every operating system has built software abstractions on top of this hardware accelerators as a middleware for multiple processor virtualization solutions. In Linux, the accelerator is named Kernel Virtual Machine (KVM). It is exposed as a character device /dev/kvm and configurable through ioctl with a large set of opcodes.

5.1.2 Executions of instructions

As seen previously, virtual machines propose emulation of hardware capacity of a server. Thus, it must emulate processor features to permit execution of an instruction stream since it can not rely on source code. One of the solution used in VMM is to translate instructions in a source instruction set architecture (ISA) to a destination ISA. Such techniques are known as *binary translation*.

5.1.2.1 Interpreter and fetch-decode-execute loop

The immediate implementation of binary translation can be provided by emulating how CPU instructions execute the associated logic. This can be achieved by looping over the instruction stream using program counter register as an iterator decode each instruction to invoke the associated logic. This approach imitates the fetch-decode-execute loop that the processor uses for interpretation. However, considering a set of instructions altogether let room for further optimizations by understanding the group of instruction logic.

5.1.2.2 Binary Translation for cross architecture support

Virtual machines typically executes instruction stream rather than instruction files which imposes constraints on how binary translation may occur. VMM uses *dynamic binary translation* where instruction stream is divided in *basic blocks* which connects with other basic blocks and form a graph known as *control flow graph*. Basic block typically begins with non-contiguous jumps in the instruction stream most commonly used in conditions and function invocations.

One of the first identified bottleneck in virtual machines was the use of code generator like qemu TCG Just-in-time compiler which is used to perform virtual machine instruction compilation into host instruction set through an intermediate representation. This technique suffers large performance penalty particularly emphasized by *code expansion* when source ISA does not fit into destination ISA.

However, a common pattern is to run guest using the same instruction set as the host machine which led to the introduction of hardware accelerator for virtualization.

5.1.2.3 Introducing the new mode

CPU vendors proposed to allow guest to directly feed instruction to the CPU hardware interpreter to speed up translations. It required a way to maintain virtual machine strong isolation, thus CPU vendors have proposed the introduction of a new CPU mode to let processor know that a virtual machine is currently running on the processor. This CPU mode is named *non-root mode* as opposed to the pre-existing *root mode*.

In x86 architecture, there are two main extensions available: AMD-V and Intel VT-X. These extensions have been extended over releases to solve performance problems of virtual machines.

5.1.2.4 Mode transitions

The introduction of this new mode permits execution of guest code on host processor. However, a subset of the available instruction set is forbidden in non-root mode for security reasons. Transitions between root mode and non-root mode on modern hardware is around 200 CPU cycles [20] on modern hardware.

5.1.2.5 A mind model of processor modes

This new mode creates a two-dimensional execution model. First dimension may represent transitions between supervisor mode (CPL 0) and user mode (CPL 3). Second dimension may represent transitions between root mode and non-root mode.

5.1.3 VM memory management

VM memory management must hide to the VM the fact that physical memory is actually virtualized.

5.1.3.1 Shadow Page Table

Before hardware support existed for VM page table management, a software technique named shadow page table was used. Shadow page tables are a hypervisor construct which enable translation of guest physical addresses (gpa) to host physical addresses (hpa). Shadow page table management relies on the hypervisor intercepting some guest events such as page invalidation, attempt to load a new page table (write to CR3) and page table mapping changes. It **write-protects** guest page tables so that any attempt to change the page table causes the hypervisor to be called to perform the modification on hypervisor page table which maps host virtual addresses to host physical addresses.

Once mappings are installed in the host, execution has almost native performance. However, on changes to page table this technique leads to lots of VMEXIT and causes TLB flushes on every exit.

5.1.3.2 Intel EPT

First releases of virtualization hardware extensions came without support for second level virtualization. Extended Page Table (EPT) [72] introduces an additional layer of page table management named *EPT paging structure* or *hypervisor page tables*. EPT can maintain transparency in guest memory management by letting a guest manage its own *guest paging structure* or guest page table. This layer implements translation of guest physical addresses (gpa) to host physical addresses (hpa) to present a x86 MMU to the guest. On Intel processors first bit of secondary VM control structure (VMCS) is used to control the use or not of Extended Page Table.

Mapping of addresses from guest virtual addresses to guest physical address relies on classical address translation described in memory management section. Here, we will focus on translation of guest physical addresses (virtual RAM) to host physical addresses (real RAM).

When accessing a guest physical address (gpa) unmapped in the EPT, an EPT violation fault is raised. EPT violation generates a VMEXIT and traps into hypervisor code to fix the fault until the execution is resumed by invocation of a VMENTRY. The extended page table directory address is stored in the host in the EPT pointer loaded in the VMCS and the CR3 register of the guest contains the physical address to the guest page table.

Extended Page Table rely on a page table translation with four levels (or optionally five). EPT page table entry uses a different layout than page table entry though some information similar information can be found like permissions (read-/write/execute), access and dirty bits.

The use of EPT enables performance gains over software shadow page table technique. EPT reduces the number of VMEXIT EPT also provides caching of gpa to hpa translation in a Translation Lookaside Buffer (TLB) which enable considerable speedups in virtual MMU translations.

5.1.4 VM IO management

There are three main classes of IO device support for virtual machines. First, VMM may provide *device emulation* where the VMM expose to the guest OS a legacy I/O device and perform I/O operations translation in VMM software to issue I/O on real hardware. This emulation technique offers limited performances. Second, VMM and VM may share a similar abstraction of IO devices for optimal communications. This technique is known as *paravirtualization* Finally, direct and exclusive assignment of hardware devices to virtual machines is possible and known as *device assignment* which we review in next paragraphs.

5.1.4.1 Device assignment

IO device attachment is a technique were physical IO devices are handed to virtual machines. However, it requires hardware support [1] to maintain isolation between virtual machines. Indeed, virtual machines may forge erroneous source

or destination addresses and read or write a physical address outside its isolation domain.

Hardware accelerator for IO operations thus propose a new hardware component known as IOMMU which provides a *DMA remapping agent* for VM isolation as well as *interrupt remapping* to accelerate interrupt delivery to virtual machines.

Single Root I/O Virtualization [135] has been introduced has an extension to device assignment by allowing multiple virtual machines to share an identical physical IO device.

5.1.4.2 Implications of device assignment for memory management

Device assignment imposes limitations on memory management. Hardware devices will perform DMA operations on host physical addresses corresponding to a virtual machine buffer. However, because of VM isolation neither the hypervisor nor the device is aware wether the buffer provided to store the result of the operation on the reception path is mapped.

There exist different approaches to deal with this problem.

A first widely used approach uses *static pinning* [162], a technique which explicitly pin the entire VM memory. If the entire memory is pinned, all DMA transactions are guaranteed to hit a mapped page. However, static pinning causes longer VM creation time.

For some time, VMs relied on *paravirtualization* to inform the hypervisor of DMA mapping. This remains unpractical when for unmodified virtual machines.

Another approach named *virtual IOMMU (vIOMMU)* [9] is an IOMMU emulation. This enables *fine-grained pining* and unpining of host pages corresponding to guest DMA addresses. vIOMMU programs physical IOMMU to only permit DMA transactions on the appropriate host pages.

More recently, some work implemented page fault capability directly into NIC controllers *NIC page faults* [92], FPGA and GPUs. However, this functionality suffers long page fault resolution delays up to hundreds of microseconds [143].

5.2 Automatic tiering of VM memory

In previous section, we have shortly presented the mechanisms used to guarantee isolation of virtual machines between each others. We have presented how virtualization extensions provided by processors helps to speed up execution of virtual machines for memory management or IO device managment. In this section, we try to determine how VMs behave on heterogeneous memory systems and in particular how automatic page placement can be performed at hypervisor level. Since VMs directly run inside processes, it could be tempting to automatically place pages in memory tiers by reusing automatic page placement techniques used for process memory management and described in section 3.5. However, as identified by vTMM [131] VMs suffer additional constraints which we review in this section. We review these constraints for each independent mechanisms that is to say first for page tracking, then page classification and finally for page migration.

5.2.1 Constraints on PT scanning

As noted by vTMM [131], classical page tracking techniques are not well suited to be used to track VM pages at hypervisor level. Raminate [69] is an EPT scanner which scans the entire extended page table for inspection of Access and Dirty bit in EPT page entry. However, full EPT scans are very long.
DAMON [116] is a Linux kernel service which monitors the access bit of EPT pages. It uses region sampling to reduce the overhead of scanning. However, it uses a static number of regions [131].
Hemem [122] which is a non-virtualized page table scanner uses Intel PEBS. However, Intel PEBS only supports host virtual addresses and not guest physical addresses.
Autotiering [85] relies on NUMA page fault tracking.

vTMM [131] relies on a CPU extension for virtualization to get a list of accessed guest pages and to prevent entire page table scans. This feature named Intel Page Modification Logging (PML) adds an entry to a page modification log after each modification to a guest-physical address that sets an EPT dirty bit. vTMM works as a kernel module which records the root address of the last level of the page table (PMD dump for 4kiB pages, PUD dump for 2MiB pages) in a local map. At the beginning of each new scan phase, vTMM module clears A/D bits of each page table entry of guest page table, then clear dirty bit in extended page table. A new scan phase is started again after a *monitoring time window*. An active page is stored in higher level pages to avoid interference on A/D bits. Promotion and demotion of levels is performed like in Linux LRU with two consecutive observation of A/D bit set or cleared respectively.

One of the problem of vTMM implementation is that they pollute A/D bit setting in the guest for page cache writeback and swap writeback. They blacklist guest page cache pages to avoid this problem which breaks transparency and still causes problem for swap support in guest OS.

5.2.2 Constraints on page classification

VM Working set estimation next requires *page classification* to separate hot pages set from cold pages set and to also assess pages access types (reads or writes). vTMM [131] observes that relying on Linux LRU is not ideal since it does more than basic access frequency counts as it also contains information on how accesses are performed [131]. Using *frequency thresholds* to determine when a page has become hot does not account for the diversity of workloads. More generally, existing page classification techniques do not differentiate read/write performances during classification.

vTMM [131] similarly to Autotiering [85] performs *access count ranking*. They maintain read/write frequency for each page and sort them by order of frequency. They introduce read and write weights to account for asymmetry of heterogeneous memory latency and bandwidth. The size of hot set and cold set is determined by the size of the underlying slow and fast memory.

5.2.3 Constraints on page migration

Traditional page migration techniques used in hypervisors would cause large overhead. Parallel copies of transparent huge pages used in Nimble and Autotiering causes slower accesses during migration. Indeed, these implementations rely on the hypervisor to unmap memory ranges before performing the page copy. Other prototypes like Hemem [122] write-protect pages before migrating them to obtain access throttling. But, write-protected pages in hypervisor causes VMEXIT which have been shown to be expensive in earlier work in shadow page table memory management for example.

vTMM [131] also relies on Intel PML to migrate pages which have been dirtied during migration. vTMM also perform parallel page migration and performs a first phase copy. After the first copy it changes the mapping of virtual addresses to the new physical address for all pages. Pages which have been reported dirty by PML log are reported in a dirty bitmap. These pages are copied again in a second phase copy. Convergence of page migration happens in two phases because pages are unmapped.

They limit the number of VMEXITs by relying on PML rather than write-protection. They show that PML based page migration and write-protected page migration (as in Hemem) perform identically and outperforms Linux page migration by a factor of 2.

5.3 Collaborative Memory Management

In section 5.1 we have reviewed internal mechanisms used by hypervisors to guarantee isolation between VMs. We have shown how hypervisors build VMs abstraction to provide the foundations for the execution of a guest operating system. VMs are executed inside hypervisors which support classic OS services such as scheduling or memory management. Thus, there exists two levels of OS services which can, in some cases, work against each other in uncollaborative ways. In section 5.2, we have discussed existing memory management techniques which try to perform best page placement decisions despite limited knowledge of guest internal memory management.

In this section, we first discuss the problem of semantic gap between guest memory management and hypervisor memory management. Then, we review proposals to use collaborative information provided by the guest to hint the hypervisor of page state and usage to prevent suboptimal decisions.

5.3.1 Semantic gap in memory management

Previous work [8, 130] have demonstrated that swapping pages in the host is inefficient. They identify multiple scenario where lack of collaboration between guest and host in the case of host swapping causes severe degradation performances.

Silent swap writes [8] occur when the guest wants to read page content of a page which is in VM disk image. It loads data in memory and the host swap-out the page.

Stale swap reads [8] happens when a guest wants to perform a disk read from its disk image to its memory. The host performs a swapin operation to bring back the destination buffer in memory. Guest performs read operation to write disk content in destination buffer.

False swap reads [8] occur when the guest overwrites a page which is not present in the host, the host will swap-in the page just for the guest to overwrite its content with its new value instead of directly writing the value on disk. This happens when the guest performs copy-on-write (COW) or page zeroing before use (after an allocation).

Decayed swap sequentially [8] happens when the host swaps out pages from the guest reading contiguous data from its disk. It causes contiguity loss which prevent prefetching.

False page anonymity [8] describes the fact that all guest pages are seen as anonymous by the host which swaps them regardless if they are file-backed or anonymous in the guest.

Dual swapping [130] occurs when a guest tries to swap a page already swapped by the host. vswapper [8] proposes a full-virtualization solution to deal with all previously stated problems.

5.3.2 Collaborative techniques

Collaborative Memory Management (CMM) [130] has entirely redesigned a memory management system which requires the guest to share memory management such as usage and residency state information with the guest. CMM maintains a state machine in the host based on information provided by the guest. They use a state machine in the hypervisor to track *usage state* (Can page content be dropped (guest clean page cache)) and *residency states* (Page Present, Page contains zero). Their collaborative solution reduces the amount of paging.

5.4 Virtual Machine Orchestration

In previous sections, we have detailed system-level mechanisms to execute VMs with a focus on memory management to improve resource usage. An alternative way of improving resource usage in datacenter relies on orchestration of VMs at datacenter scale. The understanding of orchestration solution is important to design relevant abstractions at system level. Thus, this section present basic understanding of VM orchestration which we use to carry on the discussion in the following sections. In this section, we first sketch a basic model to detail the different problems which need to be solved during VM allocation and migration. Second, we try to dig into memory waste from a VM scheduler perspective. Third, we review challenges in scheduling VMs on a set of nodes with discussion of VM allocation and VM migration where VM scheduling is involved.

5.4.1 VM allocation and migration model

One of the existing solutions to increase datacenter resource usage is to perform an optimal placement of VMs on the physical machines. One approach is to perform *static VM placing* at VM allocation time. The other approach is *dynamic VM placing* which tries to migrate VMs during their execution time.

Distributed dynamic VM consolidation can be divided into a set of 4 sub-problems [21]:

1. *Host underload detection* tries to assess when VMs should be migrated to switch the host into low-power mode.
2. *Host overload detection* tries to assess when the host is facing pressure and VMs should be migrated elsewhere.
3. *VMs migration set* tries to determine the set of VMs which should be migrated from an overloaded host to another host. It can choose VMs based on various options but usually select VMs with low activity to avoid downtime in used VMs.
4. *Destination node selection* is where the bin-packing problem is encountered. The goal is to find a server with enough free resources to start the VM. It commonly relies on allocation heuristics to guarantee that the worst allocation solution over the optimal solution is bounded by a maximum constant.

5.4.2 Understanding memory waste in the DC

Previous paragraphs have presented a basic model to understand the key challenges in VM orchestration . This model enables to understand optimization issues from a high-level perspective where allocated resources match used resources. However, at system-level VMs rely on overcommitment techniques (similar to plane overbooking) where allocated resources may be higher than available resources based on the observation that not all resources are used at the same time.

In the following paragraphs, we present Pond [95] model to understand memory waste in the datacenter. Figure 5.1 represents the two main contributions of memory waste in a datacenter: *stranded memory* and *unused memory*.

5.4.2.1 Stranded memory

Pond [95] defines *stranded memory* as a server memory leftovers following reservation of all CPU cores by virtual machines. Pond reports that stranded memory increases with the number of cores and claim an average of 25 % of stranded memory in Azure Datacenters.

5.4.2.2 Unused Memory

Pond [95] also defines *unused memory* as the difference between reserved memory resources and used memory in the VM. This metric varies over time, however, Pond reports that 50 % of VMs touch less than 50 % of their memory.

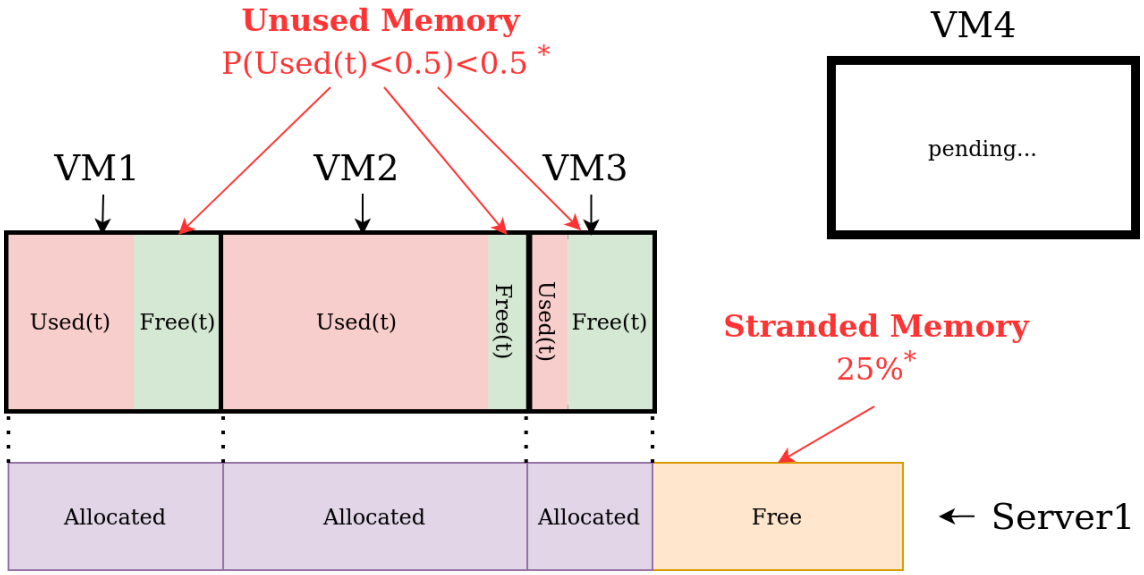


Figure 5.1: Stranded and unused memory

5.4.3 Multidimensional Bin-packing problem

As many other allocation problems, the selection of destination node sub-problem during VM migration is challenged by *external fragmentation*. That is to say a VM allocation may fail despite the sum of server free resources being greater than the allocation goal.

5.4.3.1 Bin-packing

Virtual Machine placement on a set of nodes is often referred to as VM packing. The problem of VM packing can be reduced to a multi-dimensional bin-packing problem known to be computationally NP-hard. [163].

5.4.3.2 Heuristics

Despite optimal VM placement being an NP-hard problem, it is still possible to rely on heuristics to satisfy basic requirements. These heuristics guide VM allocation to guarantee low allocation time while limiting the number of extra number of resources required compared to an optimal placement solution. For instance, *next-fit allocation* policy guarantees that VMs allocated under this policy will not use more than twice the optimal solution. Similarly, *First-fit* and *best-fit* never require more than 1.7 times more resources than the optima solution.

5.4.4 Virtual Machine Allocation

Virtual machine allocation is commonly issued by a VM scheduler like Protean [62] or Google Borg [150]. It tries to place virtual machine request once on a set of server given the knowledge of reserved resources it is aware of. VM allocator have a lot in common with job or task scheduler which are widely covered in the

literature. There are different possible architecture either using distributed agents or centralized decision-making.

5.4.5 Virtual Machine Migration

VM migration consists of moving the execution context of a VM from a source server to a destination server. VM migration is a practical feature used for *hypervisor upgrade* [110] and to reduce *datacenter fragmentation* in a similar manner to page migration (see §3.5.1).

Historically, there have been attempts to perform migration on processes directly. Although, processes are not a good unit for migration since they have hard dependencies on operating systems interfaces (system calls, ...) and shared libraries. These dependencies require exact compatible software versions running on source and destination nodes, which is infeasible in practice with different OSes distributions and software stack used. Since VMs ship a full operating system, there is no need to solve conflicts in software stacks. Instead, compatibility must be ensured on hardware devices used by the VMs. Thus, VMs are a better fit for migration.

Virtual machine migration is an important feature of virtual machines since they enable *datacenter defragmentation*.

5.4.6 VM migration techniques

An hypervisor provides an interface to migrate virtual machine atomically, i.e. it provides a roll-back solution if the migration aborts. Successful migrations results in VM being migrated from a source server to its destination server while failed migrations abort and let VM resumes its execution on the source node.

Virtual Machine migration performance can be studied according to different metrics such as *VM migration time*, *VM downtime*, *application performance degradation*, *dirty page ratio* or *VM memory size*. Google [126] claims that live migration downtime is an important factor of live migration. They report 50 ms median downtime and 300 ms 99th percentile downtime over a million live migration in their datacenter.

There exists two main implementations of live VM migration, precopy and postcopy.

5.4.6.1 precopy VM migration

The main idea of *precopy migration* [31] is to begin by copying the whole memory from the source node to the destination node of the virtual machine while source CPU runs and to copy processor state at the end. This technique works in multiple iterations, and after each iteration, the algorithm selects the set of pages which have been dirtied concurrently with the migration. This technique is implemented in qemu.

Pre-copy migration minimizes *VM downtime* and *application degradation* and also provides a clean way of *aborting the migration* [68].

One of the problem of this technique is that under high page ratio dirtying this technique shows very long migration time. Thus, this technique must rely on convergence mechanisms such as *CPU throttling* or *switching to postcopy*.

5.4.6.2 postcopy VM migration

On the contrary, *post-copy migration* [68] starts by migrating CPU and device state on destination node. Destination node then intercepts page fault and solve them by fetching pages from source node. In qemu, page fault interception is done in user-space with *userfaultd*. Ultimately, it pushes pages to speed up migration and prevent rarely accessed pages from delaying migration time. This migration algorithm minimizes network overhead by transferring pages only once. This technique is implemented in qemu.

It is common to find implementation (such as libvirt) which rely on one or two precopy phases before performing postcopy migration. This permits to speed up the migration by using precopy memory bulk transfer before falling back to postcopy for convergence.

5.4.6.3 Oasis partial migration

Oasis [173] proposes a solution to use remote memory by using VM migration technique. It is based on the principle of *partial VM migration* as opposed to pre-copy and post-copy migration which are *full VM migration*. Their approach to partial migration consists of migrating the execution context of the VM on another node and then retrieve remote pages on-demand. It differs from *post-copy live migration* by not pushing pages which have not been demanded yet. It results in the migration of the execution context of the VM and its working set and lets *cold pages* on the source node. Though, partial VM migration requires the source node to stay alive to answer memory pages requests and there could be little energy savings if the source node needs to answer multiple memory requests.

5.4.6.4 Limits

AggregateVM [30] states that relying on virtual machine migration to offer better consolidation is problematic because of **slowdown** caused by migration and the additional datacenter resource required to perform the migration.

5.5 Challenges of resource usage unpredictability

In the previous section, we have presented some of the challenges to efficient resource usage from a VM scheduler point of view. In this section we discuss how VMs resource usage unpredictability further complicates their allocation from customer point of view and scheduler point of view. First, we present the problem of ***static over-provisioning*** *of instances to avoid resource shortage. Then, we discuss* ***resource over-commitment*** *and limitation to this technique.*

5.5.1 Over-provisioning

Several studies show that a virtual machine (VM) often uses considerably less hardware resources than it has reserved [43, 123, 39, 77].

Quasar [43] shows reserved and used resources at Twitter over 30 days for an Apache Mesos [13] cluster. They show that 70% of their workloads are over-provisioned with an average of 20% CPU utilization over 60 % of CPU resources reserved on average. For memory resources, they show that an average of 40% of memory is used while roughly 80 % is reserved.

Resource central [39] shows that 60% VMs have an average CPU utilization lower than 20 %. Because average CPU utilization does not reflect CPU spikes, they observe that 40 % use under 50 % of their CPU resources to execute at 95 % of their maximum CPU utilization.

Morpheus [77], an hadoop job scheduler, analyses how their customer provision jobs. They show that 75 % of hadoop jobs are over-provisioned for a peak usage with 20 % of job over-provisioned more than 10 times. Similar they show that 90 % of jobs are over-provisioned for an average usage which also shows that provisioning based on average usage instead of peak usage would result in 15 % being under-provisioned.

5.5.2 Overcommitment and resource sharing

An existing approach to avoid wasting unused resources in a virtual machine is **overcommitment**. This technique consists on having a total allocation size larger than the available resources and thus to perform *resource sharing* of available resources among allocated resources.

5.5.2.1 High-level resource overcommitment

CPU overcommitment leverages the ability of a system to allocate more vCPUs than available CPUs on the host. CPU overcommitment is a **safe** technique since host vCPU scheduler (same as thread scheduler in qemu/KVM) can multiplex the access to compute resources without causing process crash. However, the technique hurts significantly the CPU-time available to virtual machines and leads to service level objective degradation undesirable when *minimal response time* is a desirable property.

Memory overcommitment leverages on-demand paging which lets process allocate virtual memory in larger quantity than physically available. Memory overcommitment may lead to **unsafe scenario** when no offloading device is registered and VMs device to actually use more memory than physically available. When an offloading device is registered, memory overcommitment becomes safe and commonly rely on mechanisms to maintain the virtual machine working set in local memory. However, when working set mispredict a future access, it leads to significant delays in virtual machine executions.

Cloud provider must deal with two unpredictability phenomenons. First, VM allocations exhibit patterns but remain highly unpredictable. Second, VM resource usage is unpredictable.

5.5.2.2 Unpredictable resource allocations

Protean [62] is a virtual machine allocator used at Microsoft Azure. They observe that virtual machine allocation requests vary over the week with day-time and workday having more activity than night-time and weekend days. Moreover, they observe for a single day with smaller time unit large variations in allocation requests with up to 2000 requests per second to handle during few minutes time window.

Morpheus [77] is an Apache Hadoop [12] job scheduler which tries to address the problem of variation in job execution time at Microsoft. It divides unpredictability sources of execution time of identical jobs as either *sharing-induced* or *inherent*. Sharing induced is caused by scheduler making incorrect placement decisions. Inherent sources may be caused by code changes or changes in inputs for the job. Morpheus analyses millions of job allocations over 50 thousands nodes and show that job requests exhibit periodic patterns but have a part of noise causing unpredictability. They observe potential correlation between resource sharing and job runtime.

5.5.2.3 Unpredictable resource usage

Overcommitting hardware resources is risky because the load of a VM often *unpredictably increases and decreases* for *unpredictable durations*. [77, 171, 167, 62, 98].

Morpheus [77] shows that job duration is unpredictable from a scheduler point of view which means that making assumptions on the duration of resource usage is unpredictable.

Heracles [98] is a cluster scheduler trying to reuse spare resources of latency critical services. Latency critical services such as webserver are commonly overprovisioned to meet the service level objectives. This leads to an average of 30% idle resources in google websearch servers. By sharing resources between latency critical services and best-effort tasks they manage to reach an average of 90% server utilization. They manage to maintain Service Level Objective (SLO, a service quality metric) by observing that resource sharing (CPU utilization, DRAM bandwidth) degrade performances when saturation occurs only.

Caladan [52], a CPU scheduler, observes similar phenomenon to Heracles when resources are shared. They name the slowdown resulting from resource sharing *interference*. Caladan observes that compression, compilation, spark jobs and garbage collectors work with sub-second phases of resource usage. These phases trigger abrupt resource usage changes. For example, Caladan analyses that mark phase of a GC may consume memory bandwidth which causes a colocated memcached job to have its 99th percentile latency slowed down by up to 1000 times.

Zhang et al. [171] show CPU utilization pattern in ten datacenters for servers and original applications named primary tenant. As opposed to primary tenants, they introduce secondary tenant as resource harvesting workloads which are evictable tasks trying to execute on remaining resources. They show that main workloads can exhibit either *periodic*, *unpredictable* or *constant* CPU utilization. Primary tenants exhibit an average of 85% of usage as a *constant* workload. However, on average 40 % of servers are periodic with daily and monthly period with 30 % constant usage

and 30 % unpredictable usage.

Twitter [167] performs analysis of its in-memory cache clusters by analysing the number of requests to objects and the number of accesses to objects over time. They show that requests are unpredictable with common requests spikes caused by cache systems being one of the first component behind frontend service and end-users. Moreover, they show that the number of objects accessed tends to follow the number of requests which shows that requests are not always performed on the same *hot keys*. They show that object accesses spikes are a result of traffic surges, scan access patterns, requests retry, ...

5.6 Transient Virtual Machines: Trading service level for resource usage

In section 5.4, we have seen that resource usage in the datacenter remained low because of allocation leftovers and unused resources. These resources which can be reclaimed by the cloud provider at all time are named transient resources. This section focuses on proposals to run VMs with degraded performances and availability to monetize these transient resources as much as possible. First, we present preemptible VMs which can simply kill instances carelessly to give back resources to the hypervisor. Second, we discuss approaches which try to adapt resource usage in VMs based on available resources. Third, we review harvest VMs, an approach to dynamically change VM resources by stealing unallocated resources.

5.6.1 Preemptible instances

Multiple cloud providers such as Amazon Web Services, Microsoft Azure or Google Cloud have introduced preemptible instances commonly named as Spot-VMs. This class of VMs are created on unallocated resources of cloud providers servers. Upon new allocation of traditional VMs, these spot-VMs instances may be shutdown if the VM scheduler wishes to use its resources. The idea is to sell cheaper low-availability VMs for customers while enabling cloud providers to increase resource usage.

5.6.2 Feedback control of resource usage

Brownout [86] is a software engineering model design which proposes to adapt applications resource consumption based on available resources instead of extending resources to match applications. Applications able to perform adaptation are named *brownout-compliant*. In their paper, they focus on maintaining a cloud application response time (*setpoint*) with varying compute time available for oversubscribed vCPUs (*input*). First, applications must be designed with a separation concern to split response between mandatory part and optional part. The approach will adjust resource consumption by degrading optional responses quality. Second, applications must present a *dimmer* used to affect response quality and amount of consumed resources. Third, a *controller* performs feedback control to adjust the

dimmer based on response time observations. A statistical study on an advertising system shows that a self-adaptation approach enables more robustness in case of usage spikes, hardware failures of performance interference. However, their work requires redesigning applications and some applications only implement mandatory features. Moreover, they only focus on vCPU compute resources inputs without focusing on memory resources usage.

Resource Deflation [132] tries to address the problem of transient VMs being preempted (i.e. rescheduled or killed) when more resource is required. The paper proposes another approach based on cascading resource reclamation at different levels: The paper first identifies that leveraging hypervisor resource reclamation leads to inefficient reclamation caused by hypervisor being unaware of guest memory management. Resource deflation instead proposes a cascade deflation mechanism which begins by reclaiming application resource before resorting to guest OS reclamation and finally when previous steps did not meet the expectations resort to hypervisor reclamation. They implement application deflation in memcached and JVM. Memcached deflation policy is performed by doing object eviction out of the cache by calling LRU.

5.6.3 Harvest VMs

Harvest VMs [4] is a concept recently popularized by Microsoft to increase consolidation ratio in their datacenter similarly to what Spot VM do. HVMs are collocated with regular VMs but steal their unallocated resources to dynamically adapt resource usage while offering mechanisms to give back borrowed memory.

In this section, we have introduced transient virtual machines, a set of prototypes which trades service level guarantees for higher consolidation ratio. There exists other means to achieve higher consolidation ratio notably by leveraging memory disaggregation and integrate it with orchestration.

5.7 Orchestration of disaggregated resources

Previous sections have shown how resource usage could be improved by the use of local system mechanisms to adapt resource usage (section 5.6) or datacenter reconfiguration (section 5.4). The emergence of new rack-scale interconnects and prototypes for disaggregated memory permits new scheduling opportunities by using resource leftovers not just on a local server but on remote server too. This section proposes a review of the different VMs architectures proposed to use remote resources. First, we discuss an approach based on granting exclusive ownership of remote memory pool in a rack to a single VM. Then, we review proposals to distribute the execution of a VM on resource leftovers from multiple servers. Finally, we present resource pooling, an approach to disaggregated memory where stranded memory in a rack is aggregated into a memory pool abstraction accessed using cache-coherent interconnects.

5.7.1 Memory Partitioning

Fastswap [7] and Infiniswap [61] propose partitions of remote memory by running on a server a RDMA application server to establish RDMA connection.

Infiniswap [61] evaluates cluster wide memory savings using random placement of 90 containers on 32 machines. They use 1.47 times more memory than without infiniswap which is undesirable however they achieve better memory balancing than without infiniswap where entire servers had no memory usage. [1]

Fastswap [7] retrieves disaggregation profiles (application performances degradation over varying portion of remote memory used) for various applications. They implement a scheduling algorithm with two phases. First, they check whether the job **fits** in the node by checking if the number of cores available is sufficient, then verify if enough local memory is available before falling back to checking if enough far memory is available. Second, they perform a **rebalance** phase where they assess how much far memory should the job use. Then, they require a policy to define the minimum local memory which should be used.

When accessing remote memory is required to run the job, they propose multiple policies such as **universal minimum**, a **per-job minimum** and a **memory-time minimum** which is computed based on observation of the two previous policies. The universal minimum policy is a static minimum ratio of remote memory all jobs can use. The per-job minimum defines the minimum policy statically but for each job.

Memory-time is a metric which reports how long memory is used which accounts for actual datacenter memory consumption over time. They define **local memory-time** savings as the memory-time of local memory **only** if application may perform remote accesses and if the application executes locally exclusively. They define **remote memory-time** as the memory-time of remote memory. Fastswap scheduler must find a trade-off between having more remote accesses which leads to longer local memory usage and performing every access locally which is expensive in terms of local memory used. Thus, they try to maximize the ratio of **local memory-**

[1] Their rack-scale memory usage is quite arguable because of high standard deviation in measurements and a single run for memory balancing.

time over **remote memory-time**. The ratio is maximized during execution of rebalance phase at regular time interval, and it depends on the knowledge of job execution time, the local memory ratio, the progress of job ratio. A limitation to their scheduling policy is to consider the resident set size of a process to be constant over time which is proven false.

Fastswap relies on a simulation framework to evaluate their scheduling policies. They show that using far memory can improve workload execution for jobs which are limited by available memory. They also show that upgrading available memory on a server yields higher makespan. However, local upgrades cause memory waste because additions are performed at fixed-size granularity. Moreover, for around 20% remote memory usage jobs memory-bounded jobs, jobs which perform poorly on fragmented servers see larger improvement in their makespan.

5.7.2 Distributed VM

GiantVM [170] is a recent attempt to run a single virtual machine on multiple host. GiantVM targets virtual machines with large memory capacity. Their prototype directly hijacks Linux KVM EPT page fault handler logic to implement Ivy [96] cache coherency protocol.

AggregateVM [30] is another proposal to leverage distributed shared memory abstraction to execute virtual machines. They also rely on Ivy's algorithm but exhibit better performances by implementing DSM in kernel and avoiding expensive privilege level changes. They rely on Popcorn Linux, a modified Linux kernel which runs per-CPU Linux kernel to expose a single system image.

In order to speed up their DSM, they expose a runtime NUMA topology to hint memory latencies and help allocation decisions. Moreover, they share guest page table and interrupt table location with the hypervisor to try to reduce DSM traffic on these pages. They also propose to let IO devices bypass the DSM for faster and to access exclusively local IO devices for quicker accesses. AggregateVM demonstrates 2.5 better performances than GiantVM.

AggregateVM also extend a VM scheduler to support a fallback allocation policy when a VM allocation is not possible. Thus, they implement a **best-FIFO** policy where they look for the minimal number of nodes satisfying the VM vCPU allocation constraints before starting the VM. Interestingly, they hook on VM termination to upgrade an aggregateVM to a local VM.

5.7.3 Memory Pooling

Pond [95] proposes one of the earliest models of memory savings possible with the use of CXL type-3 devices. They use stranded memory on each server to create memory pools shared by multiple sockets in the rack. Pond proposes a memory pool abstraction where memory can be hot-added and hot-removed by slices of 1 GiB to the pool with respective costs of a few µs/GB and 10-100 µs/GB. They show that sharing memory between 16 to 32 sockets would enable 10 % memory savings.

One could believe that a pool should be shared by all sockets of the cluster, but this would result in significant performance degradation. For different disaggregation

ratio ($\frac{remotememory}{totalmemory}$) of 10%, 30% and 50%, Pond observes that pools shared by more than 16 sockets enable limited memory savings. Indeed, memory savings becomes asymptotic with the number of sockets sharing the pool because VMs may only use stranded memory pools, but it is limited in using CPUs on the same machine.

Pond also relies on a machine learning model to predict the amount of untouched memory of the VM which they try to place on CXL memory. Moreover, they build a model to estimate VM slowdown depending on how much remote memory it uses (this is similar to fastswap [7] disaggregation profile) Their model uses a Random Forest classifier algorithm by collecting VM execution traces for 200 hardware counters. They try to identify which hardware counters are more relevant to predict untouched memory and VM slowdown and find that DRAM boundness alone is satisfactory though it can be improved by using other counters. They show that they are able to decently forecast VM untouched memory with an average of 25% untouched memory per VM with less than 5% over-prediction of unused memory.

In this section, we have proposed a generic overview of VMs with a review of use cases and services they offer. We have seen how these services are enforced by OS mechanisms with different solutions for instruction execution, memory management and IO device emulation. We also presented how hardware enables to accelerate the execution of VMs.

Then, we have discussed how memory management hypervisor mechanisms apply to the use of heterogeneous memory. In particular, we have reviewed solutions to offer automatic placement of pages on heterogeneous tiers at hypervisor level and we have discussed the limits of these solutions.

Next, we have reviewed how the use of two level of memory management (hypervisor and guest OS) hardens decision making for automatic page placement and how collaboration between guest and host can help achieve better performances.

The second part of this section introduces the problem of resource usage in datacenters executing VMs. We have first reviewed a basic model to understand challenges of the VM scheduler to perform optimal VM placement to reduce resource usage. Then, we explain the problem of load usage unpredictability which is the root cause of resource overuse. We have seen propositions from cloud providers to adapt resource provisioning of VMs with resource remaining and VM needs.

Finally, we have seen that new VM architectures using low-latency far memory accesses can improve resource usage by leveraging clever memory placement on the different memory tiers.

In this part, we have presented a review of academic and industry solutions to execute VMs with reduced resource consumption at datacenter scale. We first started by a review of new memory backends and their properties before showing the motivation behind new cache coherent interconnects for rack-scale computing. In particular, we have focused on CXL which has become over the last few years, the industry standard for remote memory accesses.

However, the possibility to access new remote memory hardware requires operating system level assistance to maintain or propose new memory management abstraction to applications. We have started by reviewing the core concept of memory

management in the Linux kernel to understand the implementation of legacy abstraction in a modern operating system. Then, we have discusses solutions for memory management on heterogeneous memory tiers and their applicability to disaggregated memory.

In a subsequent section, we have reviewed networking solutions for disaggregated memory with the use of RDMA, the best fabric currently available for rack-scale remote accesses. We have proposed to review mechanisms and challenges of transparent OS-level memory disaggregation prototypes and we have seen how information is semantically loss through the different layers of memory management.

We have ended our review with a description of the reasons behind resource waste in datacenters running VMs and the different proposition to solve these challenges. In particular, we have shortly presented solutions in VM scheduling which rely on live migration for continuous relocation of VMs at the cost of critical downtime. Then, we have presented server solutions to dynamically adapt resources to match actual VM usage. Finally, we have seen challenges and solutions to use remote memory in VM to fill memory leftovers at rack-scale and reduce resource waste.

Based on this review, the next chapters first present our proposal to offer transparent remote memory accesses to VMs in the rack. Second, it proposes a solution to implement fast memory elasticity in VMs and to tackle semantic gaps issues in the design of VM memory management.

Part II

Contributions

6 ODswap, transparent RDMA VM accesses

In part I, we have reviewed the various contributions related to memory disaggregation and virtualization. We have proposed a discussion which focuses on existing techniques for management of tiered memory systems and another discussion around virtual machines. However, the integration of remote memory accesses with VMs remains mostly uncovered with only very recent work focusing on that aspect. It is even more surprising since virtualization is one of the well-known use case of memory disaggregation which is expected to increase resource usage in the rack.

In this chapter, we present ODswap, a solution to offload on remote memory, a set of carefully selected pages from VM memory. Our solution tries to maintain application transparency that is to say we try to ensure that no source code changes is required in applications to support memory offloading.

In the following sections, we present ODswap in more details. First, we discuss the motivations behind ODswap. Then, we present our design and implementation proposal for ODswap. Finally, we conduct evaluation of ODswap with a study of performances and use cases made possible.

6.1 Motivation

As presented in §4.3.1, there exists multiple proposals to leverage Linux kernel swap to perform remote memory accesses using RDMA accesses. We have presented how swap supports transparent offloading of IO operations to remote memory with a selection of least-recently-used anonymous pages. Swap and kernel LRU has been a state-of-the-art mechanism for remote memory accesses in many works on disaggregated memory [57, 61, 7, 155, 100].

The support of VM execution with memory offloading techniques (e.g. swap) has been proposed in previous works [113, 29, 120, 44]. However, these prototypes still suffer multiple limitations to be adopted in a datacenter. Qazi et al. [120] only demonstrate the interest of offloading pages on remote memory over local storage by reusing existing networking storage protocols. MemX [44] proposes a very similar design to ODswap but since it does not use RDMA, it performs expensive network processing on the memory server side. Fluidmem [29] has been published concurrently to the prototyping of ODswap, and it illustrates that user-space page

fault handling (e.g. userfaultd) can deliver faster eviction and fetches of pages than swapping. However, all Fluidmem memory offloading is performed in the hypervisor and suffers from uncollaborative memory management between the guest and the host similarly to swapping. Thus, we design ODswap to propose a solution for remote memory accesses in virtual machines which is transparent to guest applications.

In this section, we review some of the limits we have identified in existing solutions which we address in our work. First, we discuss possibilities and use cases enabled by memory disaggregation for VMs orchestration. Second, we review the limits we have identified in existing prototypes which we address in ODswap. Finally, we discuss additional constraints introduced by VMs regarding memory management.

6.1.1 New possibilities enabled by memory disaggregation for VMs

Memory disaggregation is driven by different use cases such as larger memory resources for in-memory database systems (e.g. redis, memcached) or in-memory compute frameworks (e.g. Spark). Early proposals also observed an immediate potential to improve resource usage in datacenters running containers jobs or VMs. In this section, we discuss two immediate gains offered by memory disaggregation for VMs with higher consolidation ratio and reduced migration time.

6.1.1.1 Improving VM consolidation with remote memory

Based on the prior benefits drawn from storage disaggregation in datacenters, it has been identified that enabling remote memory accesses would enable to achieve higher consolidation ratio. Indeed, in all VMs architectures leveraging remote memory accesses, it is expected that the VM scheduler will have new opportunities of VM placement on stranded memory. For instance, in DSM-VMs §5.7.2 architecture stranded CPUs and stranded memory of multiple servers can be summed up to allocate a DSM-VM on the aggregated resources. In memory pooling architecture §5.7.3, a VM allocation request which needs to satisfy CPU and memory constraints needs to find a server with sufficient processors available while the memory allocation requests can be satisfied by summing up stranded memory on different servers or by allocating memory on a unique large memory server.

Thus, supporting transparent remote memory accesses will help to increase memory usage in a rack. However, accessing memory on a slower tier will degrade application performances, thus it is important to limit this overhead.

6.1.1.2 Improving migration time with remote memory

Another potential interest in the use of remote memory for virtualization is to reduce live VM migration time. Indeed, during live VM migration, VMs are executed at slower speed because of expensive page fault handling used to transparently handle the migration. VMs even end up unreachable during a downtime phase of the live migration process. Live VM migration is even slower when VM allocated memory becomes larger.

Disaggregated memory may help reduce live VM migration time by leveraging a remote memory tier which can be mapped in both the source node and the destination node.

6.1.2 Additional challenges to use disaggregated memory with VMs

We have observed two main limits to the direct use of existing RDMA swapping prototypes for remote memory accesses in VMs. First, if existing prototypes were used at guest level by passing through a RDMA network cards, virtual machine memory would be pinned which prevents VM overcommitment. Second, if existing prototypes were used in the host, hypervisor would perform uncollaborative page placement because of semantic information loss at hypervisor level. We discuss these two limitations in the next sections.

6.1.2.1 Guest swapping and the memory cost of RDMA passthrough

The use of existing remote memory swap prototypes such as Infiniswap [61] or fastswap [7] requires passing through RDMA networking cards to the guest. Indeed, RDMA network cards implement the closed-source proprietary networking stack in the hardware and there exists no RDMA emulated device for VMs so far. Thus, it is impossible to emulate RDMA NIC and leverage existing prototypes to perform remote memory accesses.

Delegation of PCIe RDMA network cards to the guest can be achieved using PCIe passthrough and SR-IOV [135].

The module in charge of supporting PCIe passthrough in Linux is *vfio* (Virtual Function I/O). It is a generic interface on top of architecture and vendor specific IOMMU drivers. In particular, it interfaces with *DMA remapping* (DMAR) and *interrupt remapping* functionalities offered by IOMMU to enable DMA in virtual machines while maintaining isolation. One of the known limit to vfio is when a VM is started with a PCIe device, vfio pins all the memory of the VM. In the following paragraphs, we dig into DMA remapping to understand the reasons for this limitation.

DMA remapping works by configuring a special page table on a IOMMU device to map IO virtual addresses (iova) on host physical addresses (hpa). In our case, IOVA can be seen as guest physical addresses (gpa). Typically, there exists one special page table for each VM. [1] This per-VM special IOMMU page table guarantees isolation between two VMs for DMA operations.

Figure 6.1 describes DMA remapping and its implication with vfio. In details, the hardware IOMMU page table is filled at hypervisor start time by vfio driver. The hypervisor vfio driver fills mapping entries to map per-VM IO virtual addresses (iova) with host physical addresses (hpa). During VM execution time, a guest posts a DMA read operation by using an iova to represent the buffer where the IO should be written by the DMA controller. The iova is transparently translated to hpa by

[1] It helps to note the similarity between IOMMU page tables for VMs and MMU page tables for processes.

the hardware IOMMU. Similarly, when a guest posts a DMA write operation, it uses an iova to represent the address of the source buffer and the hardware IOMMU translates it to hpa to actually read it.

Since vfio is not aware of which pages are used by the guest, it performs preventive *pinning* of the entire guest physical address space (e.g. all pages are mapped and present) to guarantee that to prevent DMA IO error. However, pinning VM memory prevents memory overcommitment (i.e. allocating more VM memory than hypervisor available memory). This prevents the use of overcommitment mechanisms such as ballooning which tries to reduce memory usage in each server. More generally, it prevents sharing *unused memory* in guest VMs as described in Figure 5.1.

There exists two main solutions to support fine-grained pinning of IOMMU pages which are the use of vIOMMU [143] and IOMMU page faults for unmapped pages [92]. However, vIOMMU are disabled by default in main Linux distributions and also introduce communications overhead [143]. IOMMU page faults have been shown to introduce very large overheads [92].

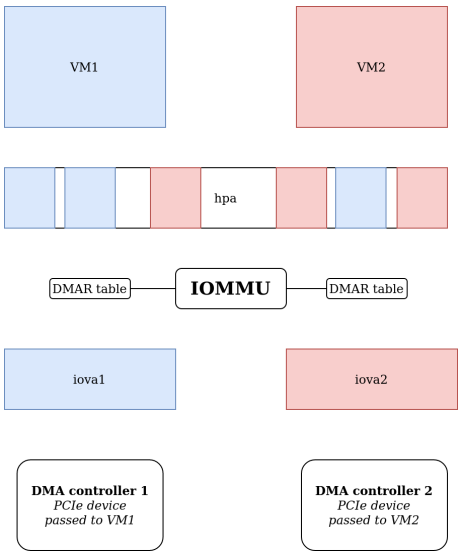


Figure 6.1: IOVA to HPA translation with IOMMU tables

6.1.2.2 Hypervisor swapping and uncollaborative swapping

In the previous paragraphs, we have reviewed how existing swap prototypes could be used in the guest and the limits to use them. Another direct approach is to directly register a swap backend in the host to perform remote memory accesses transparently. Since Linux KVM VMs are executed in processes using anonymous mappings, hypervisor swap system can directly offload VM pages on a swap backend. This solution is fully transparent for VMs, but as reported in section 5.3, it causes expensive uncollaborative page placement decisions.

Indeed, hypervisor manages a uniform anonymous view of memory and loses the information of how pages are used (e.g. IO page cache, anonymous memory, kernel memory). Thus, hypervisor swapper considers indifferently a page which backs the guest IO page cache, anonymous process memory or critical kernel memory.

This may lead to sub-optimal decisions in Linux swap system with performance degradation (e.g. swapping a clean page of the guest IO page cache, while the page may simply be discarded). We quantify the cost of these sub-optimal decisions in §7.2.2.

In this section we have presented possible use cases of disaggregated memory for VMs. In particular, we have explained how remote memory access helps to improve VM consolidation and reduce migration time of VMs. Then, we have reviewed the main issues with the use of existing swap prototypes using RDMA which are not designed for virtualization. In particular, we have explained why these prototypes can not be used at guest-level concurrently with memory overcommitment mechanisms (e.g. ballooning). We also explain why swap prototypes perform poorly when they are used in the host because of uncollaborative memory management decisions. In the next section, we discuss how these use cases and challenges have guided the design of ODswap.

6.2 Design

In section 6.1, we have discussed the use cases which make disaggregated memory a promising solution to improve VM packability in the datacenter. We have also reviewed some challenges in the design of an efficient solution to let a VM access remote memory. In particular, we have shown how existing prototypes can neither be used in the guest nor in the host to deliver efficient remote memory accesses.

In this section, we present, memory regions, the core abstraction used in ODswap memory management. The ODswap memory region abstraction is directly inspired by RDMA memory regions, and it actively relies on them. We also discuss the particularities of ODswap memory regions, and in particular how we use them to implement on-demand memory allocations and freeing of memory.

First, we present the motivation and design of on-demand memory region allocations to reduce memory consumption on memory nodes. Second, we begin by presenting how we use RDMA to support memory regions. Third, we explain how memory regions are indexed in the guest OS. Fourth, we discuss page management inside a memory region.

6.2.1 Overview of ODswap

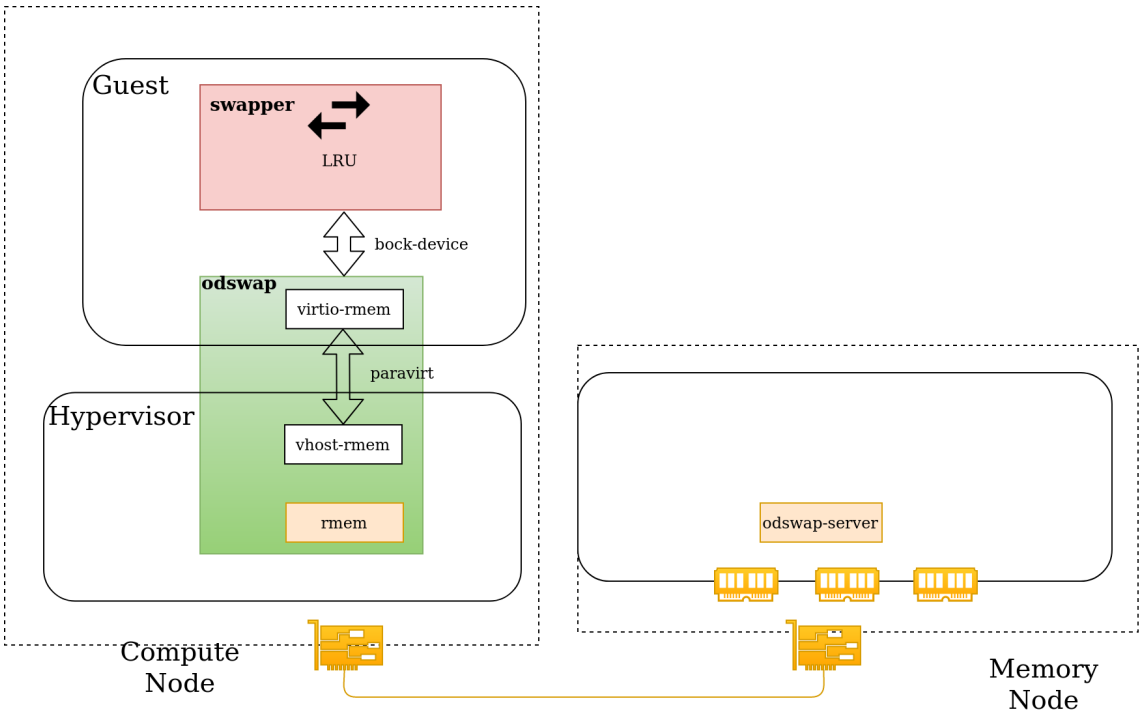


Figure 6.2: Overview of ODswap design

ODswap is made of three independent layers with different communication interfaces. The first layer of ODswap is a guest kernel module which implements the *block device interface*. It is responsible for implementing the logic behind IO operations by maintaining mapping informations. It communicates through paravirtualization with a second component loaded as a hypervisor kernel module. The second layer of

ODswap is integrated with the hypervisor compute node as a vhost kernel module. It implements the virtio server in the kernel to prevent non-scalable NIC address translations [145]. A third layer of ODswap is located on memory node and serves allocation and free requests. This is where storage of cold pages are stored.

6.2.2 On-demand memory usage

We have observed a common problem in prototypes using RDMA for remote memory accesses [7, 61]. These prototypes (infiniswap, fastswap) perform registration of a large virtual contiguous memory buffer with the RDMA card. Since RDMA registration causes all pages to be mapped and pinned at registration time that means all memory is immediately consumed. Currently, the registration operation of RDMA cards does not offer on-demand page allocation guarantees. This static usage of memory resources is incompatible with the goal of reducing memory resource usage at rack scale. Infiniswap [61] manages RDMA buffers at the granularity of 1 GiB but supports reclamation of these buffers on a persistent storage backend which makes their solution safely use memory but causes multiple accesses to be made on slow persistent storage which hurts performances. However, fastswap [7] directly allocates RDMA buffers when it starts a memory server.

The support of on-demand remote memory usage is not simply an additional feature, it is required to avoid over-consumption of memory. The implementation of on-demand remote memory usage is expected to introduce overheads which can directly impact scheduling decisions such as the decisions based on performance profiles used in fastswap [7]. In order to best use memory, we have implemented on-demand allocation for remote memory at coarse granularity. We have defined a *memory region* abstraction to describe a contiguous remote physical buffer of remote memory. This abstraction divides the swap device address space into memory regions to prevent entire pinning of the swap device address space and support distributed memory across multiple nodes. Practically, this memory region ships the start address of the remote physical buffer with the length of the region and a security key for later accesses.

6.2.3 Remote memory accesses through RDMA

Before dwelling on the details of our implementation, let us review the interest of using RDMA for remote memory. In section 4.1, we have reviewed the main benefits of RDMA over concurrent networking protocols. Thus, ODswap is uses RDMA for communications and leverages some of its advantages described in section 4.1. Notably, we use *zero-copy* and *RDMA low CPU overhead*.

6.2.3.1 RDMA zero-copy and KVM

Zero-copy communications is a technique used in networking to support communications without copying message payload in a tiered buffer. In communications with tiered buffers, the tiered buffer must be allocated in the networking stack (e.g. in the guest OS or the hypervisor). Usually, such buffers are allocated in limited

regions of memory where DMA operations can be performed. In ODswap, we are interested in zero-copy communications to avoid tiered buffer allocation to cause more pressure on page reclamation mechanisms (i.e. swap). Indeed, we expect ODswap to be used in scenario where memory is a scarce resource, thus, it is a bad idea to require additional pages allocation in the guest or the hypervisor. Both, in-kernel and in-user RDMA stacks natively support zero-copy communications. The challenge relies on enabling end-to-end zero-copy communications from the guest application to the NIC.

In qemu/KVM hypervisor, VMs are executed as qemu processes. KVM, the OS module which manages virtualization hardware acceleration, is executed in the process context of the VM (in privileged mode). This enables KVM to map VM memory and to directly read or write memory on these pages. The ability of qemu/KVM hypervisors to natively execute in the same process as the VM enables KVM to forge RDMA operations with guest application buffers with no memory copy between the VM and the hypervisor. Thus, qemu/KVM supports convenient integration with RDMA zero-copy for networking.

6.2.3.2 RDMA low CPU overhead

Another advantage of RDMA is that it supports a communication mode (i.e. one-sided operations) where remote CPU is not involved in the data path of communications. Indeed, this mode directly rely on the DMA controller of the destination node RDMA card to copy the memory page. One of the advantage of RDMA is to save CPU cycles on remote CPU which can be allocated for other tasks. Another advantage is that generating an interrupt for each page offloaded to remote memory or loaded in local memory is too expensive.

6.2.4 Live VM migration support with remote memory

In section 5.1, we have reviewed the different services offered by VMs. In particular, we have discussed the use of VMs as deployment units. We have claimed that they serve as deployment units mostly because they offer limited dependencies to other software stacks which enables easy live migration. In ODswap, we have tried to design a solution to issue remote memory accesses with RDMA while maintaining the support for live migration. One of the requirement to support live migration is to not only migrate VM memory but to also migrate the mapping information which tracks which remote memory regions are used.

ODswap guest driver maps swap sectors to memory regions to translate an IO request issued by the upper swap layer to a RDMA remote address. We have chosen to maintain the mapping information in the guest since this information is naturally migrated with the memory of the VM.

6.2.5 Implementation of device IO with in-kernel RDMA

In §4.2.1, we have reviewed some of the problems caused by management of many queue pairs and memory regions with RDMA NICs. Indeed, it has been identified that RNICs rely on multiple caches such as hardware addresses translation cache

or caches maintaining protection keys to prevent illegal RDMA operations. In our case, the address translation cache is filled by fetching hashmap entries in RDMA driver memory through DMA operations. In particular, we have seen how RNIC cache misses could multiply write latency by two when memory translation pages do not fit in the RNIC cache.

In ODswap, similarly to Lite [145], we rely on in-kernel management of RDMA communications to leverage IO accesses using physical addresses directly and thus to prevent the cost of address translation.

In user-space one-sided RDMA communications, the RDMA application configures address translation and memory protection (see §4.2.1) using the RDMA verbs API. The subset of the userspace RDMA verbs API which supports address translation and memory protection is known as *RDMA registration API* and manipulates a virtual address describing a memory buffer. Contrarily to RDMA one-sided I/O operations which are free of system-calls, RDMA registration methods directly trap in the kernel as they require assistance of kernel memory management for various tasks. First, the registration API calls kernel page pinning to ensure the memory buffer resident when the DMA operation is performed. Second, the registration API walks the page table to find out physical pages associated with the buffer. Then, the RDMA driver (e.g. mlx4) inserts new entries in the MTT hashmap for address translation, and the MPT hashmap for memory protection. These hashmaps are read-only on the hardware side, and are modified on the software side similarly to MMU page tables. For each RDMA I/O, the RNIC performs hardware lookup of hashmap entries. In ODswap, in-kernel RDMA communications directly use physical addresses to avoid hardware address translation lookups. However, ODswap still requires to call the registration API for protection to memory buffers.

In this section, we have reviewed the main design propositions behind ODswap to improve the efficiency of remote memory accesses using application-transparent guest swapping. In particular, we have presented how ODswap design builds on top of known limitations to RDMA memory management and offers on-demand remote memory allocations to reduce memory usage on remote memory. We have also presented the reasons behind in-kernel device emulation to avoid unscalable RDMA performances.

In section 6.3, we present the implementation of ODswap with a review of each component and how they communicate between each other. We also propose a micro-evaluation of RDMA and block device parameters to determine their influence on swapping performances.

6.3 Implementation

In section 6.2, we have presented the three different software components used in ODswap. These components are a guest driver in the VM, an hypervisor module which serves guest operations and a memory server where memory regions are backed and which serves memory allocation requests.

In this section, we present implementation of each of these components before proposing micro-evaluation of some of our choices. We go down through the layers of abstraction with a first focus on ODswap guest driver. Next, we present the implementation of the host driver accelerator in the hypervisor. Then, we present the implementation of the memory server to manage remote memory on the memory node. Finally, we review some of the configurable parameters through micro-optimization.

6.3.1 ODswap guest driver

In the previous section, we have reviewed the design of the memory region abstraction used to implement on-demand paging and index remote memory in the guest. In the following sections, we review the implementation of the different components with the guest driver, host driver and memory server.

This section is dedicated to the implementation of ODswap guest driver. First, we present the integration with the Linux swapper through the block device interface. Then, we present IO management with memory regions and their mapping with device sectors in the memory region tree.

6.3.1.1 Swap interfaces

In Linux, the swap memory management mechanism is responsible for the eviction of cold LRU pages to a slower backend. Linux supports *block device backends* and *file backends* for eviction of pages. It also provides an additional interface named *frontswap* which is a caching layer for an underlying block device or file backend. Thus, frontswap is stacked on top of block-device backends or file backends. In ODswap, we tried both frontswap and the block-device interface. We summarize the major differences between the two interfaces in Table 6.1. Eventually, we found that frontswap synchronous and page-per-page interface is detrimental to performances and, consequently, we rely on the block-device interface.

A swap backend is registered through swapon() system call and unregistered using swapoff() system call. There are some differences in the I/O API of swap backends especially between frontswap and block device I/O APIs. However, all backends support a common set of limited operations. They support *store/load* operations to write and read one or more pages on the backend at a specific offset on the backend. They also support, an *invalidate* operation which enables freeing the page on the backend. This operation is known under various names (discard, trim, deallocate) and it is particularly important for flash drives because these drives use a hardware copying garbage collector to collect free sectors. Thus, marking sectors as stale prevents unnecessary copies of stale data during garbage collection.

	bdev	frontswap
granularity	scatterlist	page
asynchronous support	✓	✗
synchronous support	✓	✓
size	static	determined by underlying backend
IO policy	hard-coded in driver	determined by underlying backend

Table 6.1: Features comparison between block device and frontswap

6.3.1.2 Implementing ODswap with block device interfaces

The IO stack in Linux is organized in multiple logical layers. Each layer proposes one core abstraction and provides a set of services which directly operates on it. There exist different interfaces to implement a block device in Linux. The different block-device interfaces available in Linux ship different logical layers. There exist a low-level block device interface (block_device_operations) with limited logic and a higher level interface named blk-mq (blk_mq_ops) [23] for Multi-Queue Block IO Queueing. First, the low-level layer is called for allocations of basic IO information, and, it may either decide to call the high-level layer for more advanced logic or to directly deliver the IO.

First, the low-level layer works on an abstraction named *bio*. The low-level IO layer (*bio layer*) provides a large API which supports mapping of memory buffers on device sectors, splitting and merging of bios. This layer is also responsible for stack overflow caused by IO recursion *recursion avoidance* and *queue plugging* which tries to merge sequential requests into a bigger request.

Second, the high-level layer works on an abstraction named *request* made of one or multiple bios but destined for a contiguous set of sectors. The high-level IO layer (*request layer*) is mostly known for implementing various *IO scheduling* and *elevator* algorithms. Historically, this layer relied on a single request queue, but it has recently been re-engineered to deliver higher IO parallelism by providing per-CPU queues and a queue for each device hardware queue. IO scheduling and elevator algorithms, which are mostly relevant for slow storage backend, perform *request reordering* to seek better throughput and latency, or *batching of requests* to deliver higher throughput with longer latency.

In ODswap, we rely on the high-level interface (blk-mq) and we manage the three main abstractions from low-level and high-level layers to issue IO operations which are *segments*, *bios* and *requests*. First, we manage *segments* (bio_vec) which are a physically contiguous buffer in memory identified by the first page backing the segment, a length and an offset relative to the first page. Second, we manage *bios* (block I/O), a data-structure which maps an array of segments with device sectors. It contains various additional information such as status of the I/O and potential callbacks for asynchronous completions. Third, we manage *requests* which represent a contiguous set of destination sectors. Requests carry a linked list of bios representing contiguous I/O operations. Even if a request contains multiple bios which may lead to multiple operations, it guarantees that all operations are of the same type (for example exclusively read or exclusively write).

ODswap considers two cases to deliver an operation. Either the operation fits on the destination memory region (**straightforward path**) or it overlaps multiple memory regions (**overlapping path**). In the straightforward path, the IO abstractions (segments, bios, requests) directly map with RDMA requests which requires little effort to deliver the IO. However, in the overlapping path, these IO abstractions need to be reworked mostly to split the IO to operate on non-contiguous address in remote memory. We discuss these cases in the following section.

6.3.1.3 IO management in the straightforward path and overlapping path

After discussing low-level and high-level block-device interfaces, we present the three different layers used to implement memory regions in the straightforward path. Then, we discuss IO management on non-contiguous memory regions with the overlapping path.

Straightforward path and overview of layers. As explained in §6.3.1.2, we implement the block device interface using blk-mq interface. The swap layer forges bios and encapsulates them in requests before submitting them to the IO stack. On our side, we retrieve the requests submitted to blk-mq and differentiate each read, write and discard requests based on the request opcode. Each request go through multiple layers.

ODswap *generic layer* uses sector arithmetic to compute an index to lookup the memory region in the memory region tree. This layer determines if the request overlaps multiple memory regions (see details in next paragraph named **Overlapping path**) and thus it manages *memory region vectors*. The generic layer calls the memory region tree interface, which implements on-demand allocation of remote memory regions, to *lookup-or-allocate* a memory region The memory region tree interface simply proposes insertion, lookup, states changes of memory region in the memory region tree and it is reviewed in more details in §6.3.1.5.

ODswap *high-level layer* is in charge of building *high-level requests* which encapsulates a vector of *low-level request* with one request for each memory region targeted by the IO. It also contains various information such as callbacks triggered by low-level layer, the original block layer request for notification of IO completion, a table which describes how the request spans on the different memory regions.

ODswap *low-level layer* operates on a *low-level capsule* which contains the message request payload as well as completion callbacks triggered by the host. The *low-level capsule* is used to implement IO on a single memory region. The low-level layer also implements a basic mechanism for queue overflows which feeds back messages to the queue when it is saturated.

Overlapping path. One of the problem introduced by our memory region abstraction is the possibility for an IO request to overlap two physically non-contiguous remote buffers. Indeed, as presented in previous paragraph, current kernel abstraction have always relied on the assumption that the sector address space of a device is contiguous. However, since ODswap relies on remote allocation of physical memory,

there is no guarantee that two contiguous memory regions will be backed by contiguous physical address ranges. We illustrate this problem of request overlapping in Figure 6.3 which presents how ODswap guest driver can manage non-contiguous remote memory regions.

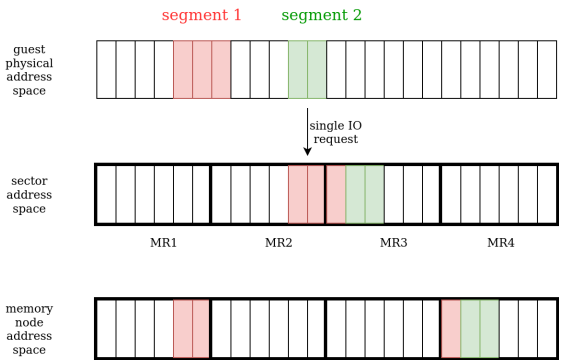


Figure 6.3: IO overlapping memory regions

Guest send requests under the form of scatterlist which are a list of segment (i.e. contiguous guest physical pages). On hypervisor side, we receive the scatterlist of guest physical addresses which we convert to host physical pages. It is important to note that this translation is possible thanks to KVM and vhost architecture because vhost has access to the page table of the process backing the virtual machine. At this stage, the hypervisor can only map source segments to a single destination segment. Thus, we introduce rmem_io_map which enables easy mapping of multiple source segments (scatterlist) to multiple destination segments (memory regions). The rmem_io_map performs automatic computation of the number of one-sided requests required to send the message. It takes into account the number of destination segments (which changes if in overlapping path or straightforward path), and also considers the hardware queue maximum size.

6.3.1.4 Creation and discarding of memory regions.

MR freeing. As seen in §6.3.1.2, the swap layer operates on contiguous sectors represented by requests. However, request sizes are not necessarily equal to memory regions sizes and requests may be unaligned with memory regions. In particular, this mismatch requires tracking each time a page is freed on a memory region in order to determine when a memory region may be discarded by ODswap guest driver. Free page tracking is implemented on each memory region at page granularity, the common granularity between IO requests and memory region.

Thus, each memory region implement freeing by maintaining a bitmap to track pages in-use. When the bitmap is clear, the memory region is freed.

MR allocation. A new memory region is allocated during the processing of an IO request issued by the swap layer. Each time an IO requests targets a sector which is part of a memory region whose state is unmapped, ODswap guest driver sends an allocation request for a new memory region. In theory, only write IO requests should trigger an MR allocation requests since the swap system is expected to read pages

it has previously written. However, in practice, read IO request with no previous writes may happen especially when the swap device is registered. At registration time of the swap device, the swapon utility typically inspects the block device sector space to detect existing file systems patterns and warn the user of potential data loss.

6.3.1.5 Memory region tree

In §6.3.1.3, we have reviewed the use of memory regions to integrate IO management with a swap device. In this section, we discuss indexation of memory regions in the memory region tree and management of concurrent operations on the memory region tree.

6.3.1.5.1 Indexing memory regions

We implement the *memory region tree* presented in §6.2.4 using a radix tree. Each memory region maintains a state among UNMAPPED, MAPPED and SENT. A memory region in UNMAPPED state means that the memory region has not been added to the indexing tree so far. A memory region in SENT state means that an allocation request for this memory region is ongoing and the IO requests should wait until the memory region is seen as MAPPED. A memory region in MAPPED state means that the memory region is currently in the indexing tree and all information are available to perform the IO.

When a write request is first initiated on a memory region which is not in the radix tree or if the memory region is in UNMAPPED state, ODswap guest driver sends an allocation request on the remote node and it adds the memory region to the radix tree and mark its state as SENT to prevent double allocation. When the guest driver receives the completion of the allocation of the memory region, it transitions the region to MAPPED state. When a memory region is invalidated, the driver sends a free request on the remote node and transition the memory region to UNMAPPED state. However, it does not remove the memory region from the radix tree to avoid undesirable contention on acquisition of the radix tree mutex. Asynchronous removable of memory regions is left for future work.

6.3.1.5.2 Concurrency on the memory region tree

There are two main scenario which can lead to concurrent insertion on the radix tree. First, concurrent insertion in the radix tree may occur in VMs with multiple NUMA node using one kswapd per NUMA node. Second, concurrent insertion may also occur during direct reclamation (i.e. reclamation during page fault handling) where multiple CPUs can initiate direct reclamation concurrently. In order to protect the radix tree from concurrent modifications, it is protected using a simple mutex as all modifications occur in sleepable context.

Apart from modifications to indexing tree, memory regions can also be directly modified when their state changes. Thus, each memory region relies on a spinlock to serialize memory region state transitions and protect them from concurrent mutations either in background reclaim or direct reclaim.

Finally, since multiple kswapd threads can modify the memory region bitmap, it is protected using a spinlock.

6.3.1.5.3 Concurrency in interrupt handling

One of the challenging code path is to insert a mapped memory region in the radix tree. In the previous paragraphs, we have seen that the memory region index tree is protected using a mutex. However, the completion of an allocation request is handled in interrupt context of the guest OS which prevents sleeping (i.e. taking a mutex lock). The challenge revolves around how to insert the memory region in the tree without taking a mutex. There are multiple solutions to this problem which can be considered. First, switching from a mutex to a spinlock would result in expensive CPU consumption to acquire the lock for threads in sleepable context (e.g. process context threads). Second, the interrupt handling routine could schedule a kernel thread in process context (i.e. sleepable) to acquire the mutex. But the cost of scheduling a kernel thread adds too much overhead.

Instead, in ODswap, we rely on a third solution which splits the insertion operation into two phases. First, we allocate the memory region structure and insert it in the radix tree with state SENT before sending the allocation request. Second, during completion handling we perform radix tree lookup, and we switch the state in the memory region to MAPPED.

ODswap guest driver uses a wait queue to block progress of write and read operations which depends on an earlier allocation.

6.3.1.6 The guest-side of virtio paravirtualization interface

In virtio, messages are delivered by the guest and acknowledged by the hypervisor in a shared memory queue (between guest and hypervisor) named *virtqueue*. Virtio supports the creation of one or multiple virtqueues for a single driver. For transmissions of messages between the guest and the host (hypercalls), virtio mainly proposes two mechanisms. First, it proposes a simple non-blocking interface to post messages along a context. Second, it proposes an additional method to trigger a notification on the host side.

Delivering a notification from an hypervisor to a guest is really similar to a device generating an interrupt to an operating system. Notifications from a guest to the hypervisor depend on the virtio communication layer used (either PCIe or MMIO). In ODswap we use a virtio PCIe device, and the guest sends the notification by using a register in the IO space of the virtio PCIe device. In the hypervisor side, the notifications triggers a callback method which consume messages from the virtqueue.

We conclude the implementation review of ODswap guest driver with the guest side of the paravirtualization interface. In §6.3.2, we resume the discussion with the host-side of the paravirtualization interface before presenting ODswap hypervisor driver for RDMA communications.

6.3.2 ODswap host driver accelerator

In §6.3.1, we have presented the implementation of the guest driver part. We have also reviewed the guest side of the paravirtualization interface to deliver messages to the hypervisor. In this section, we present how messages issued by the guest are handled in the hypervisor side.

First, we discuss the hypervisor side of the paravirtualization interface with the in-kernel acceleration of IO emulation. Second, we present how ODswap host driver manages connections between the different RDMA servers. Finally, we describe the implementation of memory region operations with independent review of the implementation of control operations and data operations.

6.3.2.1 The hypervisor-side of virtio paravirtualization interface

In §6.3.1.6, we have discussed the guest-side of the virtio communication interface. In this section, we focus on the hypervisor side of the communication interface.

Message delivery. Virtqueues propose a producer-consumer method for communication where the guest produces messages and the host consumes them. Virtqueues conveniently support **hypercalls** (guest to hypervisor messages) but are less practical to deliver **upcalls** (hypervisor to guest messages).

For hypercalls, when the hypervisor receives a message, it invokes a callback which has been registered when the virtqueue was created.

For upcalls, virtqueues are less convenient as they only support acknowledgement of requests and can not initiate a request to the guest to an unknown guest buffer. Thus, the guest must post a request message in the virtqueue before any message can be sent from the hypervisor to the guest. In the upcall path, the hypervisor can be considered as a DMA controller which only transmits information to the guest if the guest has previously posted a buffer for the hypervisor to write a message.

Notifications. Regarding notifications from the hypervisor to the guest, hypervisors can directly inject *interrupts* to the virtual machine. Thus, the hypervisor implements interrupt delivery by adding a message to the VM Advanced Programmable Interrupt Controller (APIC) to emulate an interrupt.

6.3.2.2 Acceleration of paravirtual devices with vhost

The oldest and most basic way of accessing devices in a virtual machine is to use emulated devices. Emulated devices aim at exposing to the guest an interface similar to existing hardware interfaces. Typically, an hypervisor emulates devices which have wide driver support (e.g. Intel e1000 networking cards) so that existing drivers in guest OS code base can be used in VMs. These devices rely heavily on **trap-and-emulate** where every accesses cause an exit (**trap**) from virtualized mode to the hypervisor. The hypervisor is then responsible for implementing a translation of the desired IO using its available IO interface like IO syscalls, or other storage stacks (**emulate**). However, this technique of device virtualization is sub-optimal

because trapping is expensive and the hypervisor can make no assumption about how the guest manages its device.

Paravirtualization has been proposed to address the problems of emulated device by proposing collaboration between a guest and the hypervisor to implement communications. An industry standard for paravirtualization named virtio [147] has been widely adopted across various hypervisors and OSes. The benefit of this standard is to offer portable drivers across different platforms which means reduced effort of development on the guest side to support hypervisors paravirtualized devices. Similarly, this enables hypervisors to support execution of various OSes without porting the driver code of their device in the different OS code bases.

Paravirtual devices enable higher communication throughput than emulated devices. In Linux and qemu, there exists two main class of paravirtual devices: user-space and kernel-space paravirtual devices.

Early paravirtual devices were first implemented in the hypervisor in userspace. However, these implementations suffered from expensive user-space to kernel-space transitions due to the emulation part of the device. Indeed, emulation can simply use existing OS abstractions offered by system calls to interact with the hardware. Since the hypervisor logic is mostly built in user-space, every call to OS abstractions requires an expensive privilege level change (usually one or more system calls).

Based on this observation, it has been proposed to separate the control part of device emulation from data communications. In concrete terms, the exposition of PCIe registers, device layout and the establishment of the communication path between the guest and the hypervisor is still managed by the hypervisor in userspace. However, the treatment of messages is directly handled in the kernel in order to directly call kernel mechanisms without privilege changes. In Linux, in-kernel acceleration of the datapath of virtio devices is named **vhost**.

In §6.2.5, we discuss our motivation to use vhost in our prototype.

6.3.2.3 The cluster of memory and compute nodes

RDMA provides a communication service (rdmacm) to establish connection between queue pairs of two different nodes. The main idea is that one queue pair is connected to another queue pair for communications. It relies on IP addresses provided by an implementation of IP over infiniband to bootstrap the connection phase. We skip the details of connection establishment in RDMA.

We define a session abstraction made of one o multiple queue pair part of the same protection domain. Each session represents a connection with a destination node.

6.3.2.4 Implementation of ODswap memory regions control operations

Our memory region abstraction requires communication between the compute node and the memory node for some operations such as memory region allocation, freeing. For these operations, the virtio driver has to perform an RPC on the memory node. Thus, the virtio driver uses two-sided RDMA operations. We describe the data-structures used to implement our RDMA RPC implementation.

6.3.2.4.1 RPC Ring buffer implementation

As explained previously, two-sided RDMA is more similar to traditional in-kernel TCP stacks with CPU notified when it receives a request. Since allocation and freeing requests require CPU processing, we use two-sided RDMA to be notified of the reception of allocation or freeing messages. Even if it provides a notification functionality, two-sided RDMA still requires low-level support of DMA buffers for communications. Thus, in our RPC implementation, we rely on a set of pre-registered buffers for message sending and reception to avoid the expensive cost of DMA buffer registration which requires virtual to IO virtual address translation.

ODswap RPCs build an additional queue pair fully managed in software on top of the hardware queue pair provided by RDMA. ODswap software queue pair is made of a *RX queue* for reception of message and a *TX queue* for sending of message. Each queue is implemented as a ring buffer of 4096 ring buffer entries. A ring buffer entry is composed of a pointer to hold the virtual address of the buffer, a 64 bit unsigned integer to store the physical address required for DMA operations.

In RX path, the ring buffer entry is used as a work queue element for injection of interruption in the guest out of the interrupt context as presented in §6.3.2.5.2. Ring buffer entries also contain a preallocated and DMA-mapped buffer which can host memory region information (remote physical address, length and remote security key).

In TX path, we maintain a free list of ring buffer entries with preallocated and pre-mapped payload. The payload is a fixed size 64 bit integer which encodes the remote address for freeing operation.

6.3.2.4.2 Tracking back direct allocation issuers

We use a small cache of memory region in the hypervisor to reduce the overhead of allocation in the critical path. When the cache is empty, instead of having threads wait for the cache to be filled with new MRs, they can initiates a direct allocation request which bypass the cache. One of the problem which occurs in direct allocation is when our vhost module receives a completion event for a *direct allocation* request previously posted, it has no direct way of tracking back which virtual machine initiated the request because virtual machine identifiers can not be embedded in RDMA encapsulation. Additionally, hypervisor kernel does not really hold a simple representation of unique identifier for VMs apart from process ID which requires special handling when a process terminates to avoid loss of memory regions.

In ODswap hypervisor module, we chose to deliver all memory regions to a shared pool across all VMs. Thus, ODswap uses a circular queue where it produces request during submission and consume them at completion to enforce FIFO order of direct allocation requests.

6.3.2.5 Implementation of ODswap memory regions data operations

In this section, we focus on the implementation of data operations on memory regions.

6.3.2.5.1 RDMA one-sided operations on ODswap memory regions

The implementation of remote read and write operations on memory region uses one-sided RDMA operations. Following an allocation, memory regions contain metadata information used to perform read and write operations on remote memory. ODswap splits into two modules the part responsible for handling paravirtual IO operations and the part responsible for performing RDMA communications.

The first module, responsible for paravirtual communications, retrieves IO operations from virtqueues and copy the header of the message into a dedicated kernel buffer. Then, in the payload of virtqueue elements, the module can retrieve the host virtual address pointing to the guest application buffer used for communications. The paravirtualization module uses this address to pin pages (i.e. pages are mapped and present) associated with the buffer and to map these pages in scatterlists used for RDMA communications in the second module. The pinning of pages is required by RDMA communication from the initiation of the request until its completion.

The second module, dedicated to RDMA communications, uses per-CPU queue pairs for read/write communications over RDMA (see details in §6.3.4.2). The RDMA communication module creates a single completion queue with a dedicated polling kthread to handle completion of requests. The scatterlist forged after pinning guest application pages is used to post the RDMA message to the RNIC. RDMA read-write API in Linux offers an abstraction which automatically support posting a list of memory segments and to perform the required IOMMU management for DMA operations. Thus, the RDMA communication module extends the existing RDMA kernel API to support delivery of RDMA IO on non-contiguous destination addresses. These modifications are used to reduce RDMA round-trips when guest requests are performed in the **overlapping path** (see overlapping path in §6.3.1.3).

6.3.2.5.2 Forwarding RNIC interrupts to the VM

One of the key performance issue occurs when the hypervisor tries to forward request completion to the guest. Indeed, the hypervisor receives completion in interrupt context and needs to perform up-notification to the guest by generating an IRQ. In vhost, a classical method relies on irqfd, an eventfd based signalling method between the host and the guest. However, in interrupt context the qemu process is not mapped in the kernel address space. This makes notification infeasible in interrupt context.

ODswap hypervisor modules needs a method to forward interrupts generated by the completion of a one-sided RDMA operation to the guest to terminate the IO operation. ODswap receives completion event in interrupt context and it enqueues the interrupt injection job in a work queue for deferred scheduling. It tries to batch interrupt injection by adding a linked list of allocation capsules (a capsule holds both request and completion structures). Then, once the work element has been scheduled in the host in the qemu process context, the work element loops over the list of completion events and determine the virtqueue where the request was initiated to set a bit in a virtqueue bitmap. Finally, the work job injects the request to all virtqueues which have bits set.

6.3.2.5.3 Handling concurrency between page migration and RDMA read

One of the problem which we have experienced is a race condition between live VM migration and the completion of allocation request where the allocation callback will copy the memory region in the VM. In details, after sending a request for allocation, a VM may initiate live migration and complete before the completion of the allocation request. It results in remote memory leak with a memory region allocated but the memory region metadata are lost which prevents freeing.

In order to solve the race condition presented in the previous paragraph, we rely on a technique introduced in vhost-net driver to support zero-copy [102] in VM networking devices. Thus, the hypervisor module for paravirtualization communications uses a reference counter which is atomically incremented each time an allocation request is performed and decremented when the memory region is written in guest memory. When the hypervisor module is released, it decrements the reference counter and waits in a wait queue until the reference counter reaches zero. The wait queue is woken up to test if the reference counter equals zero, each time the allocation completion callback is invoked.

In §6.3.2, we have presented the implementation of ODswap device emulation in the hypervisor. We have presented our implementation based on the in-kernel virtio server named vhost and the details for the implementation of the ODswap host-side memory region with RDMA. In particular, we have presented the implementation of RPC for control operations on memory regions with support for memory region allocations and freeing. In §6.3.3, we present ODswap memory server to present how memory regions are managed on the memory node with the server side implementation of control operations.

6.3.3 ODswap memory server

In §6.3.1 and §6.3.2 we have reviewed the two compute node components which implement remote memory accesses. In this section, we present the memory server used in the memory node.

The memory server is implemented as a kernel module responsible for serving memory regions control operations. We present the details of implementation of this module in this section. First, we discuss the implementation of ODswap memory regions control operations along the allocation of large memory buffers in the kernel to back memory regions. Second, we present the challenges of RDMA memory registration and the solution we chose.

6.3.3.1 Implementation of ODswap memory regions control operations

As presented in §6.3.2.4, memory region control operations, i.e. allocation and freeing requests, are implemented as two-sided RDMA operations. In the memory server kernel module, RDMA message reception is performed in interrupt context. It distinguishes free and allocation requests but in all cases, the implementation of server side of the request is made out of interrupt context which prevents sleeping mechanisms. Thus, we schedule control operations into a work queue executed in sleepable context.

We perform the following steps to implement memory region allocation on the memory server. We allocate the memory by getting a zeroed physical contiguous buffer from the SLAB allocator. Next, we call RDMA registration API which pins pages and pushes the buffer in the memory protection table of the RNIC. The list of allocated memory region is maintained in a radix tree index by the physical address of the memory region in order to call RDMA unregistration of the memory region and to free it. Finally, the memory server forges two-sided message with a response ID matching the allocation request ID to answer back to the compute node.

A compute node invokes ODswap memory region freeing on the compute node by giving the physical address of the memory server as an identifier of the memory region. The server implements free operation by looking up memory regions in the radix tree and unregistering memory region and freeing it.

6.3.3.2 Allocating large memory buffers

If one node is used as a memory pool, it needs to be able to allocate large memory buffers.

The problem is that kmalloc() has a limit for allocation size. Indeed, because of memory fragmentation, it becomes hard after some point in time to offer a consecutive physical memory buffer. Thus, kmalloc() limit is small for the purpose of storing multiple memory pages. Linux offers an alternative named contiguous memory allocator (CMA) and reserve at boot time a contiguous portion of physical memory which is blacklisted by the buddy allocator and managed by the CMA allocator. Though, this alternative is not very convenient because it requires rebooting when more memory is needed and the contiguous memory can not be used by the system for other purposes.

This limitation has led us to aggregate multiple memory regions of small size (A default value of 64 pages of 4096 B). It is also more convenient for on-demand allocation as detailed in section 3.4.5

6.3.3.3 Memory registration

One of the main challenge in designing efficient RDMA programs is to cleverly perform memory registration locally and deliver buffer descriptors (IO virtual address, length and security key) which can be used for communication with the other peers. Multiple works have studied memory registration architectures in the context of high-performance computing programs. There are different strategies available.

Full pinning. The first strategy is *full pinning* which consists of relying on pre-allocation and to perform memory registration on a set of buffers which will be used later. However, this approach either requires perfect knowledge of which memory will be needed in the future, or it will waste memory resources by pinning pages and preventing other tasks from using it.

Coarse grained. The second strategy is *coarse grained approach* which also relies on pre-allocation and pre-registration, but it proposes to work with static memory buffer sizes for communications. This approach removed memory registration from

the critical path and may limit the amount of communications between peers to issue requests. Thus, this approach reduces significantly physical memory consumption and still allows requests to be directly submitted without waiting for memory registration.

Fine grained. The third approach relies on *fine grained management* with one-sided communications used to communicate the size of the message to be sent and retrieving the available buffer to issue the request. This information exchange is required for every message which makes it undesirable.

Hardware page fault support. A fourth approach, recently proposed in [92] use *hardware support* to remove the burden of memory registration for DMA communications by using a special DMA page fault handler to support on-demand paging of DMA buffers. This page fault handling approach is similar to current MMU used for virtual memory translations to physical memory. This technique enables memory overcommitment between RDMA devices and other programs by issuing on-demand page allocation on first DMA accesses. However, it has been shown to hide the overhead cost in page fault management, and it is not widely available yet.

In our architecture, we rely on coarse-grained management of memory at the unit of static memory regions.

6.3.4 Late optimizations and configurable parameters

6.3.4.1 Late optimizations

In the following paragraphs, we describe late optimizations we have implemented to speed-up IO operations of our prototype.

6.3.4.1.1 Memory Region allocation cache

We identify that issuing an allocation request on a remote node on the critical path is an expensive operation bottlenecked by the page allocator on the memory node. Thus, we create two memory region allocation caches.

The first allocation cache may be shared by multiple VMs and it is located in the hypervisor module. The second allocation cache is bound to a single VM and is located in the guest kernel module.

The cache is implemented as a basic fixed-size queue protected by a spinlock. Each time a kernel thread performing a swap IO operation finds an UNMAPPED memory region, it will perform an allocation request by first query a cache of available memory regions. If the cache is full, then it can return the memory region to the guest and asynchronously issue an allocation request to the memory node. If the cache is empty, then it issues a direct allocation request to the memory node.

Upon freeing, the cache is simply refilled by appending memory region to the cache until it is full. When the cache is full, free operations are forwarded to the memory node for actual freeing of memory regions.

6.3.4.1.2 Batching

We observe that during process termination in the VM, it is common that multiple discard operation are issued on the swap device. This leads to the freeing of multiple memory regions and result in lots of messages being conveyed on the fabric. We implement a support to batch free operations by aggregating them in a single message to limit the number of round trips.

Regarding allocation requests, our current implementation supports caching of allocation operations but it does not support batching (i.e. aggregation in a single message). We do not support this feature as the existing caching layers prevent allocations to be delivered on the critical path in many cases. The other reason why we don't batch allocation requests is that it is not possible to perform each allocation in the batch concurrently. In particular, it is expected that Linux page allocator would not scale well to satisfy multiple concurrent allocations as various structure are protected by locks.

6.3.4.2 Configurable parameters

There are actually various parameters which can impact the average latency of each request. We have evaluated the following factor of influence:

1. the use of the official Mellanox infiniband verbs (named OFED) against the kernel implementation (kernel verbs)
2. the use of hardware IOMMU
3. the use of request merging
4. the allocation policy of hardware queues on queue pairs

We have observed that the use of hardware IOMMU or Mellanox OFED has little impact on the average latency. However, request merging, hardware queue on queue pair allocation policy and IO policy has real impact on the performances.

6.3.4.2.1 Allocation of hardware queues on queue pairs

One of the potential factor of performance variation is proper matching of guest block device hardware queues with virtqueues and RDMA hardware queue pairs.

Linux blk-mq system was proposed to support multi-queue devices (e.g. NVMe devices). This system proposes two level of queues: software queues and hardware queues. Depending on driver implementations, IO requests (bios) may either be submitted to the Linux block layer or bypass it to be directly handed to the device driver implementation. In ODswap, we interface with Linux block layer (blk-mq) which means that ODswap requests have previously been treated by the block layer. IO requests are first enqueued in per-CPU software queues (blk_mq_submit_bio()) by the submitting thread (swapper). Then, Linux performs different optimizations on IO requests such as request merging, or IO scheduling. Then, Linux block layer asynchronously fetches requests from software queues and sink them to one or multiple associated hardware queues. Hardware queues are managed by the block layer, however it is the responsibility of the device driver to determine how many hardware

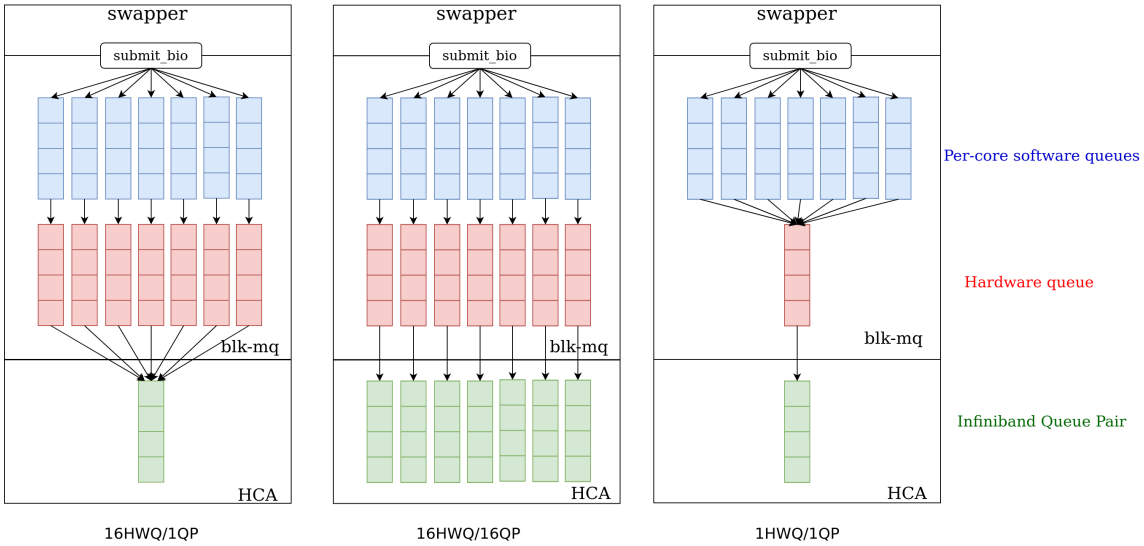


Figure 6.4: blk-mq software and hardware queues and RDMA queue pairs

queues should be created and how they are mapped with software queues and device hardware queues. Finally, the IO request is delivered to the **block driver layer** (e.g. SCSI) (queue_rq(), which is where ODswap interfaces. RDMA queue pairs are different from blk-mq hardware context queues. RDMA queues pairs are located on the RNIC and managed by the device driver.

In this micro-evaluation of an early version of our prototype, we evaluate three different configurations summarized in Figure 6.4 in a multi-thread kmeans application.

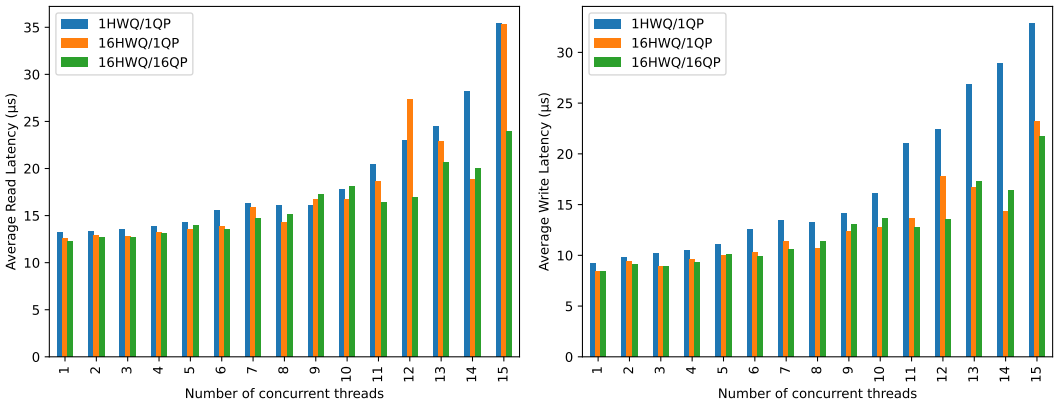
- *1HWQ/1QP* is a configuration where a single hardware queue and queue pair are created for 16 CPUs;
- *16HWQ/1QP* is a configuration where 16 hardware queues are created for a single queue pair;
- *16HWQ/16QP* is a configuration where 16 hardware queues are created for the block device while only 16 for the queue pairs.

Results are presented in Figure 6.5.

In Figure 6.5, we can see that for a single thread, all configurations perform at similar latencies. However, when the number of concurrent threads grows, average read and write latency is much higher for 1HWQ/1QP, a bit more for 16HWQ/1QP and the lowest is achieved for 16HWQ/16QP. This shows that dedicated hardware queues and queue pairs enable better performances when using concurrent applications.

6.3.4.2.2 request merging

Another potential factor of performance variation is caused by relying on *Request merging* or not. Linux commonly relies on a service named *IO scheduler* to perform request management by merging or reordering requests. IO schedulers is a very useful component for serial and high-latency backend such as HDD. With the advent



(a) Read latency (b) Write latency

Figure 6.5: Allocation of hardware queues on queue pairs

of flash drives, IO scheduler are less commonly used, and it is common to see flash drive rely on the *noop* scheduler which does not reorder requests. The noop scheduler almost provides FIFO ordering of requests with a few corner cases breaking FIFO ordering.

In our case, we focus on request merging which is a service offered by IO schedulers to aggregate together multiple bios. In multi-queue versions of the block layer, requests are enqueued on per-CPU software queues (struct blk_mq_ctx) protected by a spinlock. However, it may be interesting to merge requests together to reduce seek time on rotational drives or to avoid interrupt overheads for modern drives. Thus, in Linux v5.11.1, requests can be merged first in *plug queue* or through bio merging using *request queue*.

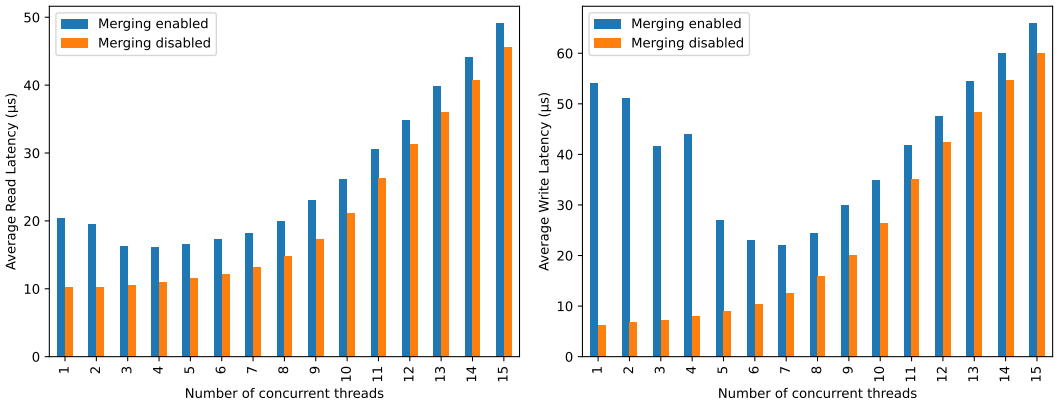
Plug merging tries to merge requests in the same request queue by holding requests during a short duration before finally flushing the plug (blk_flush_plug()) usually during kthread pre-emption through a call to schedule().

Bio merging requires taking the software queue spinlock before checking previous 8 requests in the request queue and tries to merge their bios either before or after the current request if sectors backing from the request and the bio can form a single request on a contiguous set of sectors.

In Figure 6.6, we test our bock device for two configurations: request merging enabled or disabled. We measure average read/write latency for block device IOs and we test different levels of concurrency in a kmeans application.

We can observe that enabling request merging causes read latency to double for low number of threads, but the difference soon becomes steady when the number of threads is greater than 6. Concerning average write latency, for low number of threads the latency is ten times bigger and performance difference remains higher for request merging but less significant for more than 6 threads. We observe that merging requests together adds additional latency overhead to each IO especially to write requests. However, aggregating requests yields higher request throughput but it has been shown [57] that some applications may suffer worse of throughput degradation while others would suffer more from latency degradation. Thus, request merging is an important factor of performance variation with application specific

selection required for optimal performances. In ODswap, we have chosen to activate request merging.



(a) Read latency (b) Write latency

Figure 6.6: Request merging

6.3.4.2.3 IO policies

The way the kernel issues I/O management differs a lot depending on the backend technology. Linux allows device driver developer to register the appropriate configuration for the backend used. One of such configuration is the selection of the backend as being either rotational or non-rotational.

On one hand, *rotational drives* such as HDD implement sequential I/O patterns since these drives use single head seek on a rotating disk to issue read or write operations.

On the other hand, *flash drives* usually support concurrent I/O operations, thus, it relies on per-cpu index for sector allocation to achieve better allocation parallelism thanks to partitioning of sector space.

In Figure 6.7, we have evaluated how the influence of IO policy on the execution time of a single-threaded kmeans application. We enforce with the help of memory cgroup different level of local memory based on the application resident set size. We can see that the non-rotational policy offers more than twice shorter execution time than rotational policy. As expected, the higher parallelism of non-rotational policy offers better performance gains.

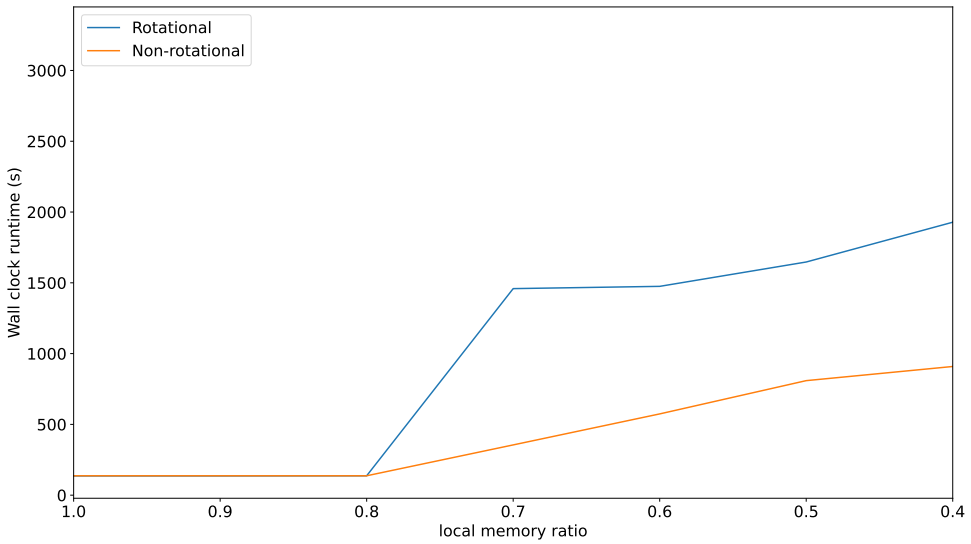


Figure 6.7: Influence of **IO policy** (rotational vs non-rotational) on wall clock runtime in single threaded kmeans

In this section, we have presented the details of the implementation of ODswap, our solution to perform transparent remote memory accesses in a virtual machine. Despite RDMA offering access to remote memory, its message-passing semantics are very different from upcoming cache-coherent interconnects which directly propose shared memory semantics.

In the next section, we propose an evaluation of ODswap.

6.4 Evaluation

In section 6.3, we have reviewed the different challenges and solutions to implement ODswap. In this section, we propose to study the performance impact of using transparent remote memory with ODswap. First, we try to assess perform degradation in various applications based on the amount of remote memory used over the total amount of memory normally required. Second, we try to determine how application performance degradation can be modelled with hypervisor knowledge to help VM scheduling issue relevant placing to limit performance impact. Third, we compare ODswap with other comparable prototypes which can be used in VMs to access remote memory. Fourth, we try to determine the performance of application in DSM-VMs architectures (§5.7.2) and memory pooling architectures (§5.7.3) to determine potential limits and advantages in each architecture. Fifth, we try to assess how ODswap helps to handle sudden memory usage in VMs compared to other solutions. This evaluation tries to understand if swap-based prototypes are a good fit to implement elasticity in VMs with low response time while maintaining good application performances.

Across the following sections, we run the applications described in Table 6.2. This table presents the different applications based on the number of threads used by the application and the *average resident set size* value we have measured for each application.

Application	#threads	AppRSS	Description
quicksort	1	8 GiB	C++ quicksort on integers
k-means	16	8 GiB	Python scikit [117] k-means
pagerank	16	8.6 GiB	Spark [168] GraphX pagerank on wikipedia categories subgraph
Alternating Least Square (ALS)	16	21.2 GiB	Spark [168] ML recommendation algorithm on MovieLens dataset

Table 6.2: Summary of applications used in our evaluation

6.4.1 Disaggregation profiles

*We present per-application **disaggregation profiles** which enables to track the impact of performance degradation. The study of such profiles is not new and has already been conducted by Gao et al. [57], Fastswap [7] and Infiniswap [61]. Disaggregation profiles offer interesting simulation perspectives, and they have been used by Gao et al. [57] to forecast the performance degradation of applications with remote memory. Moreover, these profiles can be used, as in Fastswap [7], to propose container scheduling policies which try to balance performance degradation and resource usage based on these profiles. In the following section, we plot the disaggregation ratio for a set of applications using ODswap.*

Description. In this experiment, we try to plot, for each application, the performance degradation of applications for different local memory ratio (i.e. ratio of local

memory usage over application resident set size). Local memory ratio is modified across runs by changing VM memory size to the desired local memory ratio based on the measured average resident set size of each application. In our experiment, we try to enforce an intended memory ratio in guest applications comprised between 0.5 and 1 with increments of 0.1. We have observed that there exists a difference between the measured local memory ratio of a guest application and the intended memory ratio. In our plots, local memory ratio always refer to the measured local memory ratio. We refer to the plots obtained as **disaggregation profiles**.

Application details. *quicksort* performs multiple runs through integer ranges with uniform page access frequencies meaning that the working set of pages is roughly equivalent to the resident set of pages.

ALS runs a matrix factorization algorithm on MovieLens dataset which is bottlenecked by IO operations (load/writeback) on dataset pages.

pagerank runs pagerank algorithm implemented in Apache GraphX library on wikipedia category graph. Pagerank measures the importance of each vertex in a weighted graph by running a *graph walk* through all vertices in the graph and update their state until it reaches a stationary configuration. Thus, the access pattern depends on the underlying graph and in the case of Wikipedia graph, some nodes will be hotter than others with clusters vertices which will have higher access frequencies. Additionally to the application pattern, pagerank is a Java application and runs in a Java virtual machine, which regularly runs a garbage collector. In pagerank, we have observed around 170 such garbage collection phases (with G1 algorithm). Since garbage collector are known to have poor locality mostly caused by heap walk, we could expect high performance degradation even with a small local memory ratio.

kmeans run Elkan algorithm [48] for kmeans and iterates through uniform ranges of objects in memory. We have observed large allocations and expensive zeroing to initialize the lower bounds used to represent the distance nodes and all centroids. Similarly to java (described for pagerank), python uses a garbage collector which could also degrade performances. Internally, python uses a generational garbage collector which rely on three generations of objects: newly created objects, first-cleaning survivors, second-cleaning survivors. We have observed in kmeans 176 collections in first generation, 15 in second generation and 1 in last generation.

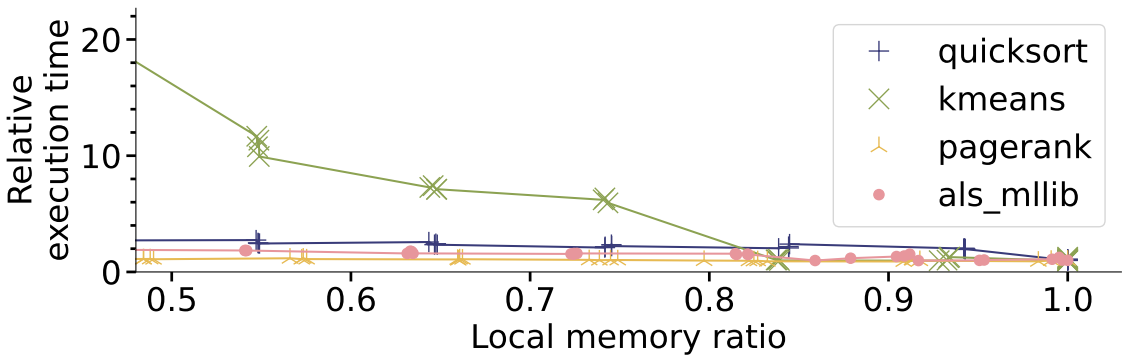


Figure 6.8: Disaggregation profiles of different applications

Observation. Figure 6.8 shows that *pagerank* suffers steady little performance degradation with execution time never bigger than twice the optimal configuration. *ALS-MLLib* has no performance degradation when more than 85% of application memory is local but execution time is twice longer when the portion of local memory is comprised between 85% and 50%. *quicksort* disaggregation profile has a similar curve shape than ALS with no performance degradation when performances with 95% local memory and a steady 2.5 overhead between 50% and 95%. *kmeans* disaggregation profile first shows no performance degradation when local memory is higher than 85% and a very high execution time overhead which goes from 7 at 75% of local memory to 10 for 55% local memory ratio.

Interpretation. In this experiment, we have identified that each application has different memory access patterns (e.g. sequential or random) with different access frequency for each page (e.g. uniform accesses or limited working set). Generally, when the working set (the set of frequently accessed pages) does not fit in local memory, application performances quickly degrades. However, frequency of page accesses remains very relative as some applications may have a rather uniform frequency of page accesses.

In this evaluation, we have shown that applications exhibit very different performance variations depending on the amount of memory maintained locally. Disaggregation profiles are important figures for VM scheduling since they directly exhibit the trade-off between application performances and resource usage.

Disaggregation profiles have been used in previous works. In particular, Gao et al. [57] have studied the impact of latency and throughput on disaggregation profiles to determine general acceptable latency and throughput values for remote memory accesses. However, disaggregation profiles are mostly relevant in a context where customers may report to the execution platform the job it is going to execute. Next section discusses blind performance degradation for efficient VM scheduling.

6.4.2 Comparison with other backends

Existing storage disaggregation prototypes can also be used as swap backends to extend VM local memory. Storage disaggregation prototypes are production-ready with strong engineering efforts, open-source implementations and no unexpected behaviours. In this section, we try to compare how these prototypes perform compared to ODswap.

6.4.2.0.1 Description. In this experiment, we try to determine the performance of our prototype using different cloud applications for various backends. In particular, we compare ODswap with:

1. *lNVME*: a local NVMe drive exposed in PCIe passthrough with vFIO
2. *rRAM*: a remote ramdisk that is accessed through the NVMeoF protocol
3. *rNVMe*: a remote NVMe drive that is accessed through the NVMeoF protocol

In each backend and for all applications, we run the VMs with 50% of the average resident set size as local memory and the rest as remote memory accessed through swap. For example, in *quicksort* application, the VM executes with 4 GiB of local memory and 4 GiB of remote memory. For each application, we compute an average execution time when the VM is overprovisionned in memory which corresponds to a setting where local memory ratio is above 100%. We report the normalized execution time of the applications on each backend.

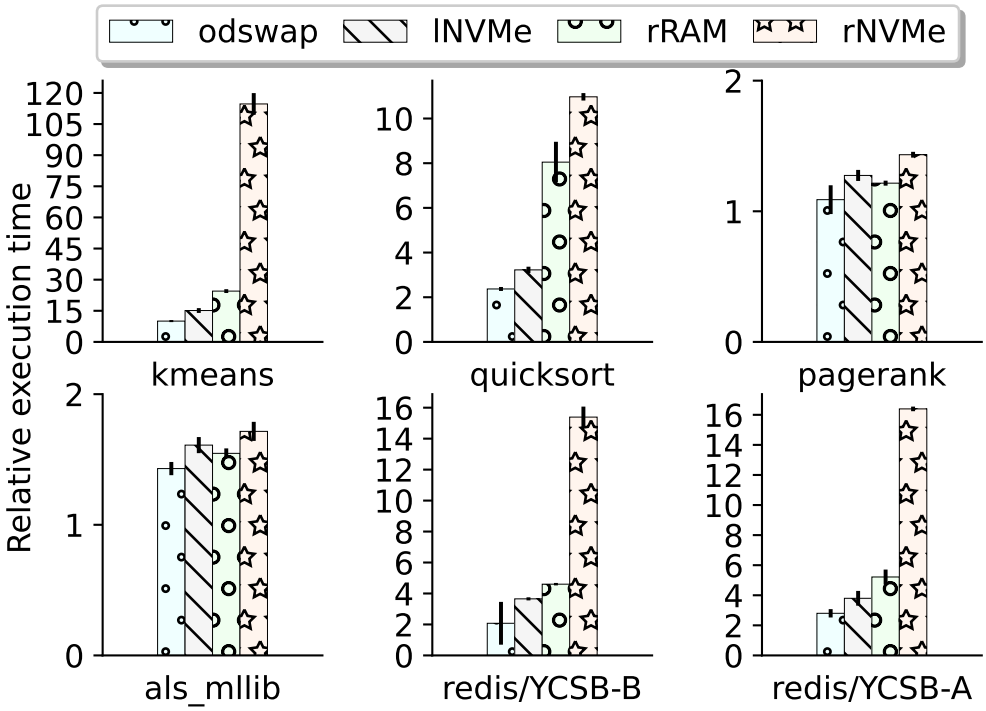


Figure 6.9: Disaggregation profiles of various applications

Observation. In Figure 6.9, we can observe that for each application ODswap outperforms all the other prototypes. lNVMe and rRAM backends exhibit comparable execution time. However, the rNVME backend leads to very long execution time which can be up to 10 times slower (kmeans). Furthermore, we can see that performance degradation for similar usage of local memory changes significantly across applications with 10 times slower execution for kmeans with ODswap and 1.5 times slow down for ALS-MLlib with the same backend.

Interpretation. First, it is important to note that ODswap compares with the advantages of rNVMe and rRAM which permits remote accesses to a storage or memory backend while lNVMe on the contrary can only support accesses to a local storage NVMe drive. Second, ODswap achieves significant acceleration for different applications compared to other implementation. We identify that ODswap acceleration is caused by the use of one-sided RDMA with coarse-grained allocations.

Indeed, rRAM and rNVMe rely on NVMeOF which internally uses fine-grained remote backend allocations by using RDMA two-sided messages and perform the IO using one-sided operations. This causes a RDMA RPC to be issued for each IO operation while ODswap only requires a RDMA RPC for the allocation of a new memory region.

In this section, we have presented the performance advantages of using ODswap other concurrent alternatives. We have noted differences in performance degradation for a same backend depending on the application in-use. Thus, in next section we study in more details the evolution of application performances depending on the local memory usage.

6.4.3 Comparing DSM VMs and disaggregated VMs

In section 5.7, we have discussed the different architectures which use remote memory accesses to improve packability of VMs in a rack. In this section, we propose to review the performance impact of distributed VM architectures (see details in §5.7.2) and VMs performing remote memory accesses through a writeback cache layer (ODswap). We review why the prototypes offer similar approaches and can be fairly compared together before presenting the advantages and drawbacks of each architectures.

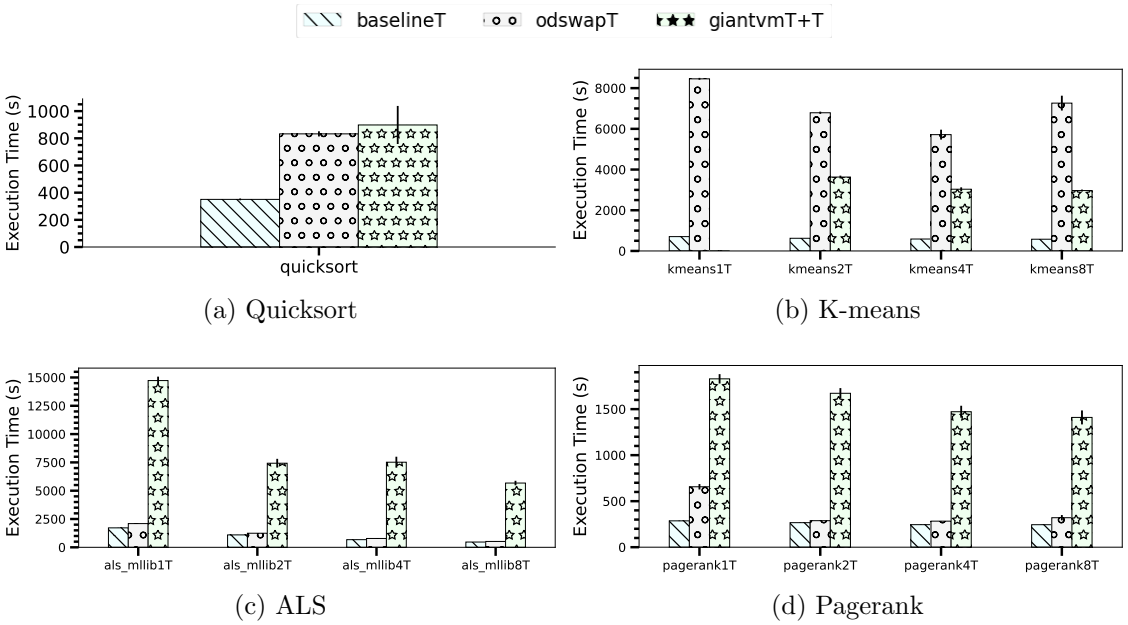
Description. In this experiment, we try to run a fair comparison of the performance of applications between a DSM-VM executing on two servers and a disaggregated VM. DSM-VMs offer better perspective of resource usage since it enables to use remote processing units, not just remote memory. We study performance degradation with the number of threads to exhibit the limits of DSM-VMs with contention. In order to have a fair comparison we compare applications using ODswap with T threads while the DSM-VMs use T threads on each server (i.e. $T + T$). Indeed, this thread configuration offers a similar complexity for the VM scheduler since finding T unallocated CPUs on a single server is always simpler than GiantVM configuration which requires the VM scheduler to find two server with T unallocated CPUs. In ODswap, we select a local memory ratio of 50 %. We present our results in Figure 6.10.

Observation. First, in Figure 6.10, we observe similar performance degradation between ODswap and GiantVM on the single threaded quicksort application. However, execution of the quicksort with an unmodified VM is twice faster than for ODswap and giantVM.

Second, for kmeans, we observe that giantVM outperforms ODswap by a factor of two.

Third, for ALS, we observe similar execution time in ODswap and in baseline around 1000 s while giantVM is 15 times slower with $15,000$s to execute.

Fourth, for pagerank, we observe that ODswap execution time is close to the baseline, however, giantVM execution time is between 4 to 6 times slower.



(a) Quicksort (b) K-means

(c) ALS (d) Pagerank

Figure 6.10: Comparison of various application execution time on GiantVM and ODswap with the number of threads

Interpretation. Both DSM-VMs and ODswap have similar latencies to fetch a page from a remote node. However, in DSM-VMs, CPUs from different nodes access a shared memory layer which aggregates memory of each node. Thus, DSM-VMs implement hypervisor-level cache coherency protocol which can be expensive in applications heavily sharing memory as in pagerank and ALS which run garbage collection algorithms. BLABLA * Both DSM-VMs and odswap have similar latencies to fetch a page from a remote node. However, in DSM-VMs, CPUs from different nodes access a shared memory layer which aggregates memory of each node. Thus, DSM-VMs implement hypervisor-level cache coherency protocol which is expensive. -¿ il faut ajouter "parfois meilleur, parfois pire, ça dépend du pattern d'accès mémoire : pour kmeans, peu de partage entre thread =¿ deux fois plus de threads permet à giant-vm d'être meilleurs, pour ALS et pagerank, beaucoup d'accès aux mêmes pages à partir des threads sur différentes machines =¿ odswap est meilleur"

This section has proposed an architectural comparison of far memory accesses between distributed shared memory (DSM) and disaggregated memory. While some applications perform better on DSM-VMs, the cost of software cache coherency in the DSM can be extreme for some workloads (up to 15 *times slower). Based on these observations, we pretend that disaggregated memory architectures in VMs are more relevant for many workloads than distributed memory architectures.*

One of the other use case which motivated our work with ODswap is the goal to absorb peak usage of memory in a VM without termination of the VM. In §6.4.4, we describe our experiment showing the behavior of ODswap when memory resource become scarce in the VM.

6.4.4 Handling peak usage

We have seen in section 5.5 that resource usage in a VM is unpredictable and this makes adaptation of resource provisioning complex. We have seen that this leads to user statically provisioning their instance for the worst resource consumption scenario. One of the possible solution offered by ODswap to limit overprovisioning is to access remote memory resources which where unallocated at VM start time.

In this section, we study the performance impact of ODswap and concurrent prototypes to use resources at execution time. In this study, we consider two different scenarios. The first scenario is a smooth memory increase which simulates an increasing memory consumption in a VM which grows higher than the memory resources initially provisioned. The second scenario considers a sudden memory increase in the hypervisor. Such a scenario is a typical representation of the allocation of a new VM on the same hypervisor.

In both scenario, we try to simulate mechanisms available to cloud providers to increase memory usage in addition with devices support remote memory accesses. We run VMs with a balloon driver to support lending unused memory across VMs. We also monitor if applications experience memory shortage by monitoring the number of page fault per time unit (page fault rate). When memory becomes scarce, both in the VM and in the hypervisor, we initiate a live migration to another machine to support execution of the VM with its entire memory capacity.

6.4.4.1 Smooth increase in memory usage.

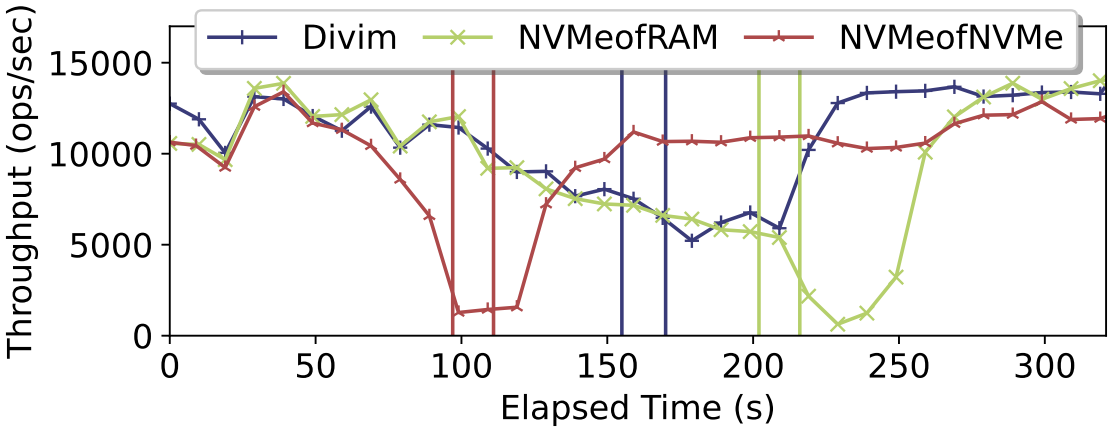


Figure 6.11: Smooth spike in memory usage in YCSB-load/Redis

Description. In Figure 6.11.a, we simulate a smooth increase in memory usage. On a machine with 48 GiB of memory, we start Redis in a VM with 32 GiB of memory, while another VM already consumes 32 GiB and runs stress-ng. In order to simulate scarce memory for the Redis VM, we load a balloon driver in the Redis VM but not in the stress-ng VM. After the boot of the Redis VM, we start the YCSB load phase on another machine, which smoothly increases the memory pressure of the Redis VM as new keys are added. Our benchmarking script also implement

a method used in some VM orchestration framework to detect when VM memory becomes rare. We rely on a threshold method which initiate the migration of the VM when the page fault rate grows higher than a limit.

Observation. At 75s, ODswap starts to offload memory. During this phase, the throughput decreases from 12k to 6kops/s. At 150s, ODswap migrates the VM because the page fault rate reaches the threshold (first vertical bar). After the migration (second vertical bar), the throughput reaches its maximum in 50s. We observe that the performance degradation remains decent (minimum of 6kops/s) despite the stop-the-world phase caused by migration. This result shows that, thanks to ODswap, we can aggressively use memory overcommitment to pack more VMs on the same physical machine: the performance of the VM remains acceptable, even if the host runs out of memory.

NVMeofRAM behaves almost like ODswap, except its throughput collapses (minimum of 1kops/s). Moreover, NVMeofRAM consumes 32 GiB of local and 32 GiB of remote memory, which makes the comparison with ODswap unfair since ODswap consumes at most 32 GiB in total. With NVMeofNVMe, the throughput collapses (minimum of 1.5kops/s) and remains lower than 6kops/s during roughly 50s.

Interpretation. While storage techniques (NVMeofRAM and NVMeofNVMe) makes the VM unusable with YCSB throughput reduced to 13% of peak throughput, ODswap is able to limit performance degradation and maintains execution around 50% of peak throughput.

6.4.4.2 Sudden increase in memory usage

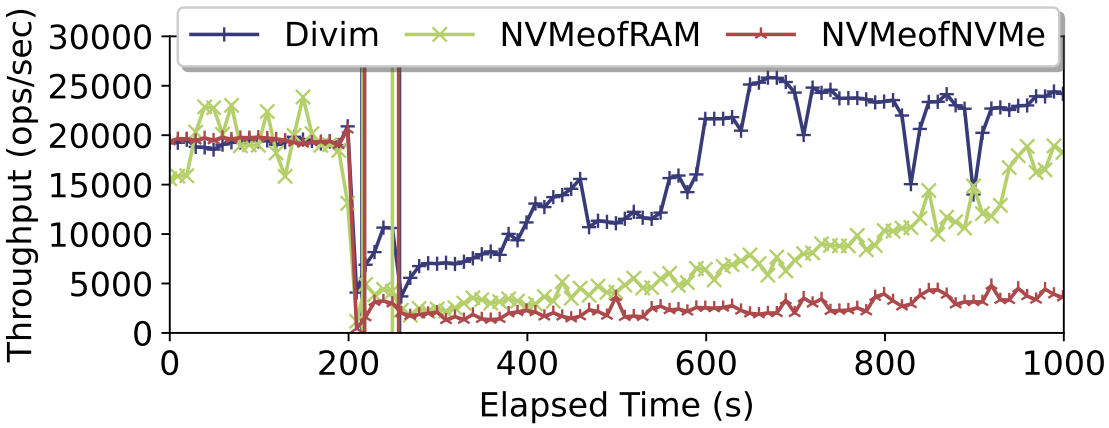


Figure 6.12: Sudden spike in memory usage in YCSB-A/Redis

Description. In Figure 6.11.b, we simulate a sudden spike in memory usage. We report the throughput of the run phase of YCSB-A (50% read, 50% update) in a VM of 32 GiB that executes Redis. At the beginning of the run (up to 200s), Redis runs alone and uses the 32 GiB.

We simulate a sudden spike in another VM, which leads to instantaneous loss of 8 GiB in the Redis VM at 200s. At that time, the VM offloads immediately 8 GiB on machine M. Taking 8 GiB from a VM is unlikely in production, except if we imagine a Spot VM scenario. Spot VMs are VMs without service level agreement that can be killed if a non-spot VM needs resources (see section 5.6). With ODswap, instead of killing the Spot VM, we can reclaim its memory and use the page fault rate to determine if the VM should be migrated.

Observation. When the VM loses its memory, ODswap throughput decreases quickly to 5 kops/s because of intensive memory offloading. NVMeofRAM and NVMeofNVMe throughput drops to around 1 kops/s when memory becomes scarce. Then, the hypervisor detects that the page fault rate reaches the threshold and our benchmarking program triggers the migration of the VM.

When the VM restarts on machine B, since ODswap uses a prepost algorithm (see §5.4.6.1), the VM is slowed down for two reasons: (i) the post copy phase of Qemu loads the pages that are not yet migrated from machine A, and (ii) the memory offloader of ODswap swaps in the offloaded pages from machine M. We observe that ODswap recovers its initial throughput of 20kops/s after 350s, against 2250s for NVMeofNVMe and 950s for NVMeofRAM.

Interpretation. Overall, this experiment shows that application running on ODswap remains usable almost all the time (throughput higher than 5kops/s), while this is not the case with NVMeofNVMe. This result shows that disaggregated VMs are a promising technique to implement spot VMs. With ODswap, spot VMs remain responsive, and they are not killed when another VM consumes the hardware resources.

In this section, we have performed evaluation on a set of application for different configurations. We have shown that each application suffer different performance penalty and we have characterized this degradation using disaggregation profiles. Then, we have compared ODswap with different backends and notably backends which are used to support storage disaggregation. Next, we have reviewed the differences between DSM-VMs and disaggregated VMs for different applications and we have shown that disaggregation offers advantages for some applications by saving cache coherency messages. Finally, we have studied how ODswap reacts to peak memory usage with smooth demand and sudden demand.

This chapter introduces our first prototype, ODswap, a system to transparently let applications running in VMs use remote memory. ODswap directly leverage the swap interface of the guest OS to sink IOs in paravirtualization queues treated by an in-kernel middleware to convert them to RDMA operations on remote memory.

In details, our main contributions compared to existing prototypes are the following.

First, our proposal is designed to support transparent remote memory accesses for VMs efficiently. In particular, we avoid the problem of ***uncollaborative swapping*** *(see §5.3.1) if remote memory accesses were performed through hypervisor-level swapping.*

Second, we implement a paravirtualization server in the host kernel to avoid ***privilege level changes*** *and* ***PCIe DMA address translations****.*

Third, we noticed that multiple prototypes actually reserve large static partitions of remote memory with expensive cost on remote server. We propose an alternative approach to concurrent prototypes swapping on entirely pinned remote memory partitions. In ODswap approach, remote memory is allocated on-demand and invalidated after being used.

During the evaluation phase of this prototype, we have observed large performance degradation in many applications for remote memory accesses using a swap device even for ODswap. Indeed, we have measured that remote memory accesses with ODswap for 4 kiB pages are much longer (approx. 30 μ s) than raw RDMA accesses at the same granularity (approx. 2 μ s). Concurrently to this work, various papers [125, 103, 28] have reviewed the various reasons behind the overhead of swap-based techniques (see §4.3.4). RDMA remains a viable solution to support disaggregated memory in language runtimes [155, 125] and databases [27]. However, recent prototypes [104, 95] rather prely on existing NUMA machines to simulate remote memory accesses on a cache coherent interconnect such as CXL.

Based on these lessons, we have tried to move away from systematic page fault handling techniques for the design of our next contribution. Additionally, the review of concurrent prototypes supporting memory overcommitment of virtual machines has drawn our attention towards other interesting challenges which should be addressed for memory disaggregation support. Next chapter proposes a discussion around our observation and our current proposal.

7

Motivating hypervisor and VM co-design

In the previous chapter, we have reviewed the main problems and contributions identified in the literature to manage heterogeneous memory systems and to manage virtual machines with dynamic resource usage. In this chapter we focus on the identification of main limits to existing VM memory tiering systems and VM memory adaptation.

Notably, in section 5.5, we have pointed out studies which show that virtual machines resource usage changes unpredictably over time. We have reviewed some of the solution proposed in the literature to adapt VM memory at runtime. Thus, the first section of this chapter proposes an in-depth review of production-ready systems to dynamically adapt VM resources with an emphasis on response time.

A common pattern reported in multiple works presented in section 5.3 is the information loss across the various layers of memory management known as the semantic gap.

In §4.3.4.1, we have discussed how the semantic gap between language and kernel leads to ***IO amplification*** *caused by a mismatch between object size granularity at language level and page size granularity at kernel level. This has already been widely studied in the context of remote memory accesses by AIFM [125]. In this chapter, we rather focus on studying* ***hypervisor level semantic gaps*** *in existing tiered systems and upcoming tiering systems like CXL. We focus on the study of performance degradation introduced by virtualization to access remote memory. First, we focus on page metadata information loss (dirtiness, file-backed or anonymous mappings) between guest OS and hypervisor in tiered-memory VMs. Existing work have reported how this information may lead to* ***uncooperative swapping*** *[8] scenarios illustrated by hypervisor swapping. Then, we study another known hypervisor semantic gap which is* ***topology exposition****. Lack of hardware informations (memory throughput, memory latency) at guest level can lead to suboptimal page placement. We also conduct a review of performance degradation when resorting to hypervisor level page placement.*

7.1 Memory overcommitment techniques

As seen previously, a key motivation for datacenters is to run more VMs on fewer servers. One of the solution is to support ***memory overcommitment****, a situation where the sum of allocated VMs memory is bigger than the server memory. However, achieving memory overcommitment with low impact on system performances is hard especially when all VMs require memory at the same time. One of the problem memory overcommitment is the lack of a fast abstraction to grow and shrink VMs memory resources dynamically.*

Traditionally, the hypervisor has mostly considered VMs as static instances using boot-time memory limits. Interestingly, the use of boot-time limits in VMs is not imposed by the hypervisor. It is rather imposed by a static guest OS memory management which struggles to give back resources when it uses them. Moreover, this boot-time provisioning combined with the unpredictability of resource consumption in VMs (section 5.5) leads users to over-provision their VMs for future peak resource usage.

In this section we study the performance of existing solutions to dynamic memory usage in VMs. These dynamic solutions either proposes to lend part of their boot-time allocated memory to another guest (ballooning) or to rely on the ability of the guest OS to add physical memory resources (memory hotplug) at the cost of expensive and sometimes infeasible defragmentation operation. Moreover, most solutions are managed by the hypervisor which requires monitoring resource usage before issuing configuration changes in feedback control systems with low reactivity. In this section, we propose a detailed analysis of time-contributions for the most common memory mechanisms to run dynamic VMs.

7.1.1 Memory Ballooning

Memory ballooning [153] is a technique introduced with first SMP hypervisors which proposes a solution to dynamically change VM size. virtio-balloon [147] is the standard ballooning driver supported by multiple hypervisors. It relies on virtio transport protocol [147] implemented over PCIe or MMIO for communications between guest and host. virtio-balloon is only a mechanism for reservation of memory in the guest. It does not implement policies regarding when memory should be reserved or how much should be saved. The implementation of policies is left to an hypervisor component known as auto-ballooning which performs calls to the hypervisor side of the balloon API.

The balloon API is the following:

```
//Grow balloon by size i.e. shrink VM usable memory by size
void inflate(size_t size)

//Shrink balloon by size i.e. grow VM usable memory by size
void deflate(size_t size)

//Report current balloon size
size_t get_balloon_size(void)
```

7.1.1.1 Known performance issues of classic memory ballooning

Recent studies [66, 67] have reported multiple major problems in the implementation and design of memory ballooning such as *lack of NUMA awareness* and *lack of Huge Page support*. Such limitations include design or implementation limits caused by the dependency on Linux page allocator for balloon inflation. Indeed, virtio-balloon inflation does not take into account vNUMA topology and consider uniform guest topology. In some cases, it may result in VMs performing remote memory accesses.

Second, it relies on page allocator which works at 4 kiB as a default granularity and may fail to perform transparent huge page allocation of 2 MiB or 1 GiB.

Existing reviews focus on application performance following ballooning commands however one of the limits to dynamically changing VM memory remains mechanism speed. Thus, we propose in this section a review of bottlenecks in virtio-balloon with a focus on the speed of each command of this mechanism.

7.1.1.2 Memory ballooning speed

We begin our study of memory ballooning with an evaluation of the speed of VM memory capacity growth and shrink. Indeed, modifying dynamically VM capacity is harder when changes are slow as these changes will cause longer execution of guest application with low resource capacity.

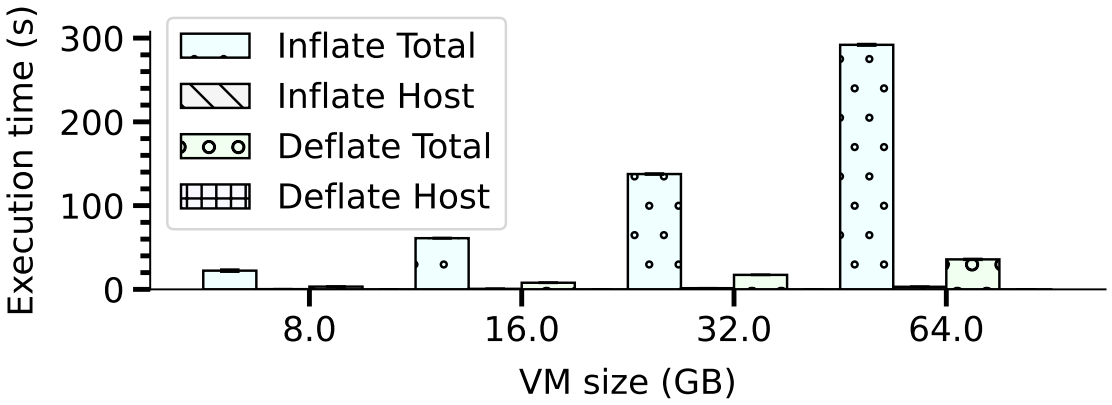


Figure 7.1: inflate and deflate time

Description. In Figure 7.1, we start a 16 vCPUs virtual machine running Linux kernel 5.11.1 with a maximum memory capacity of 8, 16, 32, 64 GiB. The virtual machine begins with an empty balloon letting the guest with 8, 16, 32, 64 GiB of available memory. The experiment is made of two steps: First, we perform a balloon inflation of 5, 13, 29, 61 GiB letting 3 GiB of usable guest memory. Second, we perform a balloon deflation of 5, 13, 29, 61 GiB to let the guest run with 64 of memory again.

Observation. Figure 7.1 reports the execution time as a function of virtual machine size (in GiB). We can see that the duration of the inflate operation with guest

and host time contributions goes from 22 s for a 5 GiB shrink to 290 s for a 61 GiB shrink. The total duration of the deflate operation goes from 3 s for 5 GiB memory increase to 35 s for 61 GiB capacity increase.

Interpretation. First, we can directly infer than both inflate and deflate delays grow linearly with the size of memory change. Second, for all sizes, inflate operation, which shrinks VM memory capacity, is longer than the deflate operation. Finally, from this experiment, we directly observe long delays for simple capacity changes in an unstressed guest. These long delays are to slow to absorb spike memory usage and will trigger memory reclamation in the guest. The delays reported in this experiment are lower bound and can be even longer if guest CPU time must be shared between applications, memory reclamation operations and memory ballooning.

7.1.1.3 Analysis of bottlenecks

Figure 7 [1] and Figure 9 reports CPU time contribution of qemu after issuing respectively inflate and deflate request of 61 GiB in a 64 GiB VM.

For each command we collect traces of CPU time contributions in the userspace qemu process and KVM accelerator module, and in the guest OS.

	Memory remove **inflate**	Memory add **deflate**
Hypervisor		
Host-to-guest request	9.3 %	20.7 %
EPT violation & MMIO emulation	18.6 %	3.5 %
madvise DONT_NEED → TLB shootdown → MMU notifier → syscall overhead	31.2 % → 10.2 % → 10.2 % → 10.8 %	None
madvise WILL_NEED → syscall overhead → Page Walk, swapin	None	19.6 % → 14.8 % → 4.8 %
others (function calls, privilege changes)	25.7 %	39.2 %
Guest		
Guest-to-host request → MMIO write → notification	8 % → 2.77 % → 5.0 %	15.4 % → 7.3 % → 8.1 %
allocation	7.1 %	0
freeing	0	1.4 %
Total (number of CPU cycles)	339 783 248 027	126 548 509 066

Table 7.1: Overall time contributions for virtio-balloon

[1] All icycle graphs are provided in the appendices for the sake of clarity

In Table 7.1, we provide a summary of the time contribution of each sub-mechanism involved in the processing of the command either from hypervisor point of view or guest point of view. The details of the time contribution of memory ballooning are presented in chapter 8.3.2.1. The details of the communications overhead between guest and hypervisors are not presented in the table, however, we have observed multiple round-trips between the guest and the host to perform inflate and deflate operations. In our experiment we have observed that a request to modify (grow/shrink) VM size by ±61 GiB issues 62,464 MMIO writes and notifications. This large number of round-trips explains the heavy cost in MMIO writes.

7.1.1.4 Automatic Ballooning: Feedback Control Overhead

Previous sections on memory ballooning only illustrates the problems of this technique as an isolated mechanism. In this section, we review the additional overheads introduced by feedback control.

PVE [119] automatic ballooning algorithm is a closed-loop feedback control algorithm. It defines **host used memory** as a *process variable* [2] and **VM memory capacity** as a *set point* [3]. The algorithm tries to maintain **host used memory** to 80% of host total memory to prevent the activation of reclamation mechanisms at hypervisor level. The algorithm acts on VM balloon size to shrink or grow usable memory per VM to balance between satisfying the two constraints:

$$C1 : \forall t, used_{host}(t) < 0.8 \times max_{host} \tag{7.1}$$

$$C2 : \forall t, \sum_{VM} curr_{VM}(t) = min(\sum_{VM} max_{VM}, 0.8 max_{host}) \tag{7.2}$$

where: $used$ = Currently used memory in the VM
$curr(t)$ = Current size of a VM at time t
max = Maximum memory size of a VM

In the following experiment, we try to understand how much overhead feedback control adds to the existing delays of memory ballooning.

Description By default, PVE uses a sampling rate Δt of $10s$ with maximum variation ΔM on the set point of 100 MiB. In Figure 7.2 we configure automatic ballooning algorithm to use different value of Δt and ΔM. We start the experiment by inflating the balloon in a VM i.e. by shrinking its available memory so that the VM is left with 8 GiB of memory out of its maximum of 32 GiB of memory. Then, we activate the PVE automatic ballooning algorithm, and we report for each different value the time taken by the automatic ballooning algorithm to converge towards its stable state where the VM is left with 32 GiB of memory.

[2] measured value in control theory
[3] configurable parameter in control theory

Observation For PVE default automatic ballooning values ($\Delta t = 10s$ and $\Delta M = 100MiB$), we can observe that the algorithm is not able to reach its stable state. As shown in Figure 7.2, for more aggressive values with $\Delta t \to 0$ and $\Delta M >> 1$, the response time is much shorter and ends up being limited by ballooning speed evaluated earlier. Moreover, when we perform monitoring of CPU usage during the execution of the control loop algorithm, we observe that the control loop program does not impose a large overhead on CPU consumption however one of the cores running the VM is consuming 100% CPU cycles.

Interpretation This experiment presents the additional time contributions introduced by feedback control algorithms for automatic changes in VM memory capacities. There exists in control theory a known trade-off between accuracy and speed. This result states that increasing control frequency causes, on one hand, faster response time but, on the other, it is also responsible for lower accuracy to determine the desired memory of the guest. This is caused by aggregating sampled data in a tight time interval for decision-making which may fail to capture overall tendency of a change. Moreover, any misprediction by feedback control of the guest behaviour will generate undesired CPU usage in guests which hurts the performance of user workloads.

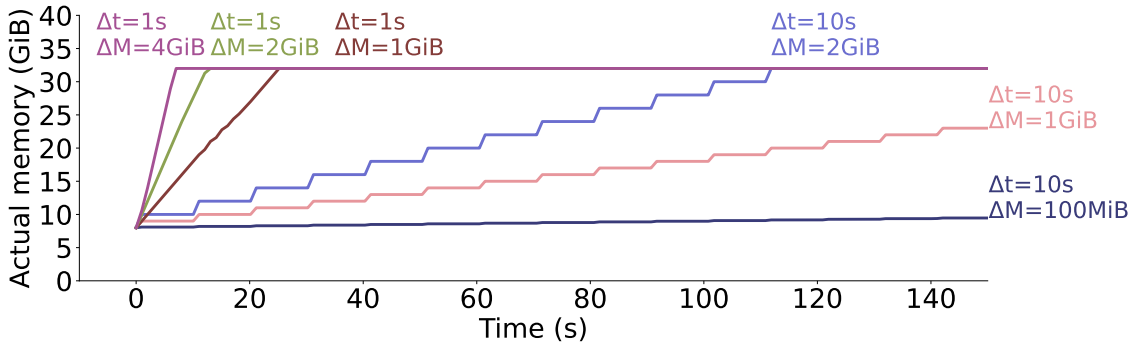


Figure 7.2: Reactivity of Proxmox auto-ballooning algorithm in a scenario requiring deflation

7.1.2 Free-page-reporting: A guest-initiated technique

Free page reporting (FPR) [158] is a recent paravirtualization feature introduced to achieve convergence of guest OS used memory and hypervisor used memory.

7.1.2.1 Free-page-reporting, guest-driven hypervisor page freeing

Free page reporting is added to virtio-balloon on both hypervisor and guest OS side. FPR is composed of two components: an extension to virtio-balloon paravirtualization layer and a client-side implementation of page reporting mechanism. First, FPR reuses the existing paravirtualization communication layer of virtio-balloon

and the exposed PCIe device. It adds a new dedicated communication queue isolated from traditional ballooning. Second, FPR subscribes to receive freed pages reported by a guest kernel mechanism named *page reporting*.

Page reporting [47] is integrated with Linux page allocator. When reported free pages reaches page_reporting_order watermark, it enqueues in a work queue a job to free multiple pages. The work is scheduled within 2 seconds. It issues a *page reporting cycle* for each zone and for each page order comprised between 0 and 10. The page reporting cycle first performs a *fill phase* to create a scatterlist of pages by looping through buddy allocator free lists to isolate unreported free pages. Then, it issues the *report phase* by calling the FPR report callback which sends pages stored in the scatterlist over the FPR paravirtualization queue. The *drain phase* removes pages from the scatterlist and reinsert them in the buddy free lists with reported pages tagged with a new page flag (PG_Reported). This flag is used in next iteration of the cycle to track free pages which have already been reported. This helps to avoid reporting them multiple times.

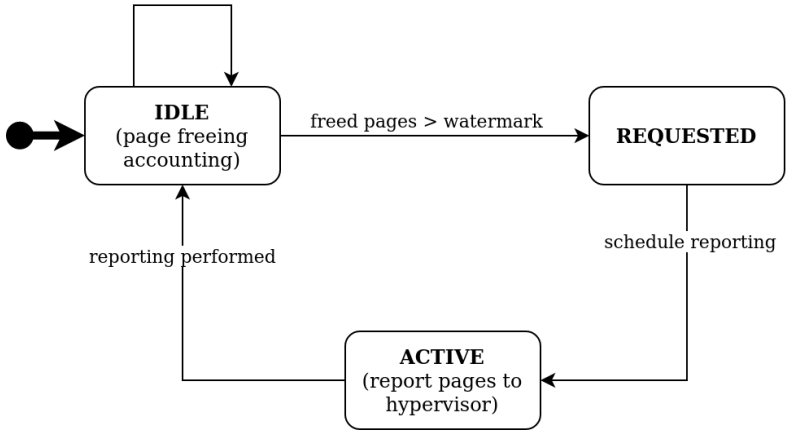


Figure 7.3: Free page reporting global state machine

As shown in Figure 7.3 FPR maintains a global instance (struct page_reporting_dev_info) which tracks the state of the page reporting mechanism. Thus, FPR defines three states: *idle* when no reporting is being done, *requested* and *active*. Initial state is IDLE. State changes from IDLE to REQUESTED when page reporting wants to free some pages after crossing the watermark, from REQUESTED to ACTIVE when the reporting job is scheduled, from ACTIVE to REQUESTED when another cycling request is made because of excessive freeing pressure while the first request is not entirely processed. Finally, state transitions back from ACTIVE to IDLE after pages have been reported to the hypervisor.

On hypervisor side, qemu calls madvise() with DONT_NEED flag on reported pages to let the hypervisor OS reclaim these pages. Free-page-reporting communicates free pages to the hypervisor using a *free page block*. The default free page block order is pageblock_order i.e. 2^{10} pages meaning 4 MiB for 4 kiB pages.

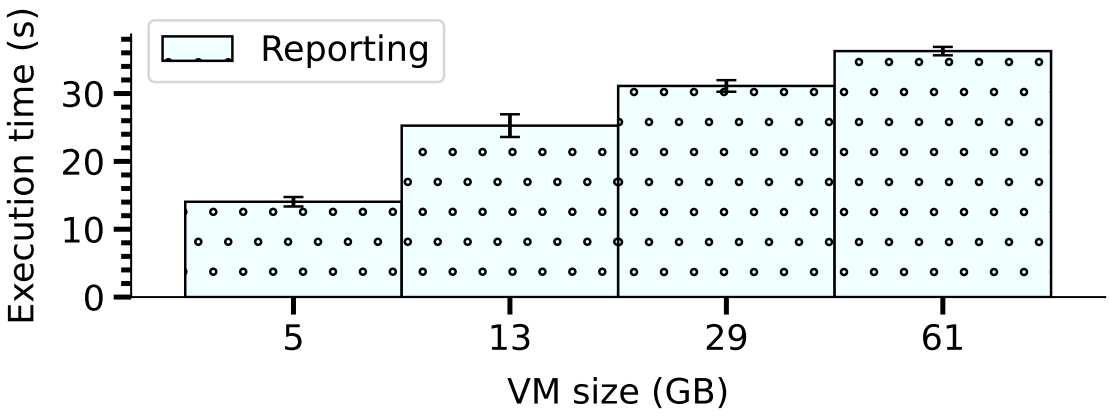


Figure 7.4: Free-page-reporting latency depending on page size

7.1.2.2 Analysis of bottlenecks

Similarly to traditional ballooning, we try to assess how quick is free-page-reporting to perform page freeing depending on the size it tries to free. Figure 7.4 starts a virtual machine with 8, 16, 32, 64 GiB size. Then, a memory stress job simulated by stress-ng tool is ran during 30 s and uses 90 % of the VM allocated memory. The 10 % leftovers are left unallocated to avoid waking up OOM process killer. It performs anonymous memory allocation only. After termination of the process, pages are freed, and page reporting is initiated. We start recording elapsed time after process terminates and finish recording after the VM used memory drops under 3 GiB.

7.1.2.2.1 Bottlenecks

On the hypervisor-side, FPR spends most of its time (97%) in zap_page_range() following madvise(MADV_DONTNEED). However, in free-page-reporting, **MMU notifier invalidation** which ensure coherence of EPT and hypervisor page table accounts for 80% of the reporting time. MMU notifier invalidation time is further split into 3.7 % for TLB flush of EPT entries on all CPUs (KVM_REQ_TLB_FLUSH). The other MMU notifier invalidation time is spent which accounts for 95 % is spent in kvm_unmap_hva_range to unmap the host virtual address space ranges of reported pages. The high cost of this method comes from removing entries in rmap, a structure mapping gpa to shadow page table entries (spte).

Apart from MMU notifier invalidation, time left is spent in TLB shootdown of the hypervisor page table (hva to hpa page table) with 13% of total reporting time.

7.1.2.3 Reactivity of free page reporting

In the following experiment, we try to determine how fast the VM used memory (RSS) follows the actual memory consumption of processes inside the VM.

Description. In this experiment, we run a VM with 32 GiB of memory with free-page-reporting. After 30 s execution, a memory intensive job (stress-ng) is started in the VM to consume 28 GiB of memory during 60 s.

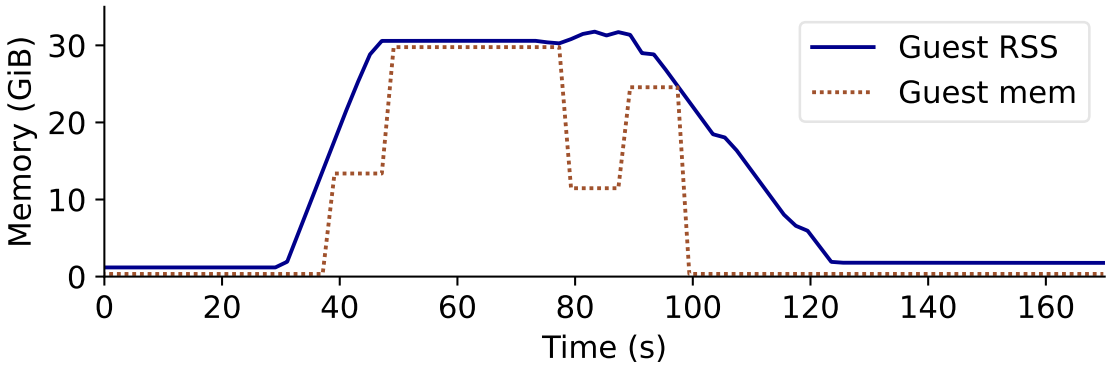


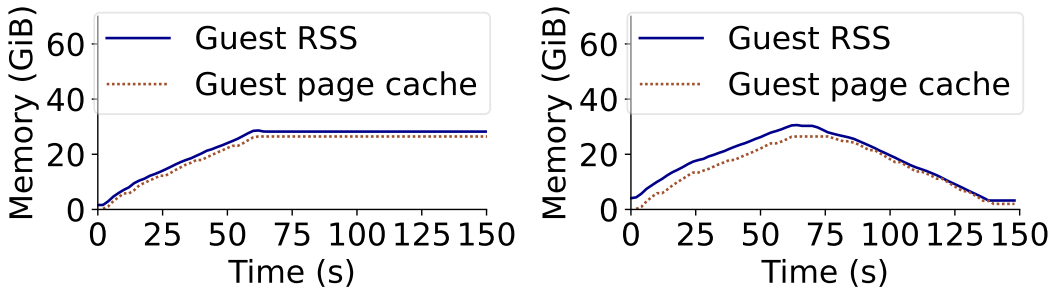
Figure 7.5: Free page reporting reactivity

Observation. In Figure 7.5, around 90 s, a monitoring job in the VM observes that pages have been freed and we can observe that the reporting process leading to freeing pages at hypervisor level completes 30 s later.

Interpretation. This experiment proves that FPR is able to successfully lower the memory footprint of a VM by freeing its pages at hypervisor level. However, the technique is not very fast for freeing pages as variation of 32 GiB of memory still require 30 s to complete. Since FPR is guest-initiated, there is no additional overhead required by hypervisor-level feedback control algorithm.

7.1.2.4 Behaviour with an IO page cache

In this section, we try to illustrate some of the design limitation of FPR for applications heavily relying on the page cache.



(a) free page reporting

(b) ballooning

Figure 7.6: Ballooning vs. free page reporting with an IO cache

Description. For this experiment we create a 32 GiB VM and we run an IO job using 28 GiB of memory with 12 workers for 60 s. In FPR, we let guest OS report unused pages while for ballooning we run an inflate command in the guest.

Interpretation As shown in Figure 7.6, FPR is unable to reduce VM used size contrarily to ballooning. Indeed, ballooning can trigger page reclamation mechanisms while FPR is unable to perform it and will let the page cache grow.

7.1.3 Memory Hotplug

Previously, we have reviewed performance problems of memory ballooning and feedback control algorithms for memory ballooning. Another widely used technique to dynamically adapt a VM memory is memory hotplug. Recent works [54, 67] have rebooted this technology for resource usage gains in the datacenter.

7.1.3.1 Transparent Memory Hotplug

Transparent memory hotplug is the historical memory hotplug technique. It leverages ACPI to inform the guest of changes in the physical memory configuration. It works by adding/removing DIMM to a guest with granularity of 128 MiB [66] at least. Traditional memory hotplug is NUMA-aware. However, it usually requires hot-unplug of memory DIMMs that have been previously plugged [67]. Traditional memory hotplug is considered as offering limited opportunities for DIMM hot-unplug compared to a newer approach named virtio-mem.

7.1.3.2 Paravirtualized Memory Hotplug

Paravirtualized memory hotplug or virtio-mem [67, 66, 146] is a recent paravirtualized device introduced in qemu and Linux to tackle problems in memory ballooning and legacy memory hotplug. Each virtio-mem PCIe device is seen as a contiguous physical address space range in the guest physical address space. Upon each request reception, a subrange is allocated out of the device range. Devices are divided into memory blocks (e.g. 2048 kiB) which can be plugged or unplugged.

virtio-mem defines three different granularities and supports four actions. The different granularities are:

1. *big-blocks (BB)* are the memory unit to hot-plug and hot-unplug memory between the guest OS and the hypervisor;
2. *Linux memory blocks or sections (MB)*, which are the granularity to which memory is added to the page allocator (System RAM);
3. *sub-blocks (SB)*, which are a logical granularity for logical plug and unplug operations in the guest OS only.

The four supported operations are:

1. *memory hotplug*, which makes a new DIMM device visible to the OS;
2. *memory hotunplug*, which performs the reverse operation;
3. *memory onlining*, which adds memory to the buddy page allocator;
4. *memory offlining* which removes memory from the buddy page allocator.

Memory hotplug and hotunplug operation will perform fake onlining and fake offlining of memory to whitelist and blacklist memory part of memory DIMMs. These logical operations are referred to as fake-onlining and fake-offlining.

Hotplug and hotunplug operations are serialized using a mutex. A page flag (PG_offline) is used to mark pages logically offlined.

Memory pages part of guest OS ZONE_NORMAL comes with little warranty to be hotunplugged. In particular, unmovable pages are caused by kernel allocations. Unmovable kernel memory typically includes the memory map (array of struct page), page tables and SLAB. On the contrary, user-space processes which mostly rely on page cache or anonymous pages rely on movable memory. Thus, a new memory zone named ZONE_MOVABLE has been introduced as a zone for user-space processes page allocation. Memory hot-unplug can aim for memory hot-unplug of blocks in this MOVABLE zone. Despite the existence of movable zone, some pages remain unmovable because of corner cases of memory management such as userspace requiring page pinning.

However, the existence of ZONE_MOVABLE requires careful balancing of memory zones between NORMAL and MOVABLE to avoid a scenario where the kernel is left with no memory for unmovable allocations. (A common ratio of MOVABLE:NORMAL=4:1 is used)

Similarly to ballooning, virtio-mem requires the hypervisor to issue hotplug and hotunplug requests. Thus, virtio-mem requires creation of virtio-mem PCIe device at boot time, bound to a NUMA node. Hypervisor can then send configuration requests to the device to update a *requested size* value which

virtio-mem suffers the same problems as auto-ballooning by trying to balance host memory resource across VMs with inherent feedback control response time.

7.1.3.3 High-level speed analysis

In this section, we try to evaluate the speed of hotplug and hotunplug mechanisms under different scenarios. We test the following three configurations:

(a) *No stress job* is a configuration where the VM is idle from its allocation to its termination

(b) *After stress job* is a configuration where the VM hotplugs memory, run a stress job until completion and tries to unplug memory.

(c) *Concurrent stress job* is a configuration where the VM concurrently tries to unplug memory while unplugging.

In Figure 7.7, we evaluate *no stress job configuration* by starting a 64 GiB and 16 vCPUs VM and plug the following sizes: 8, 16, 32, 64 GiB. In *no stress job* configuration, we can see that hotplug delay is negligible with a few microseconds overhead only and thus it does not appear on the plot. However, hotunplug delays grow linearly with the memory size removed and goes from 5 s for 8 GiB to 36 s for 64 GiB.

In Figure 7.8, we evaluate *after stress job* and *concurrent stress job* to determine if memory hot-removal suffers additional overhead after an application has made extensive usage of plugged memory. In these configurations, we start a 64 GiB and

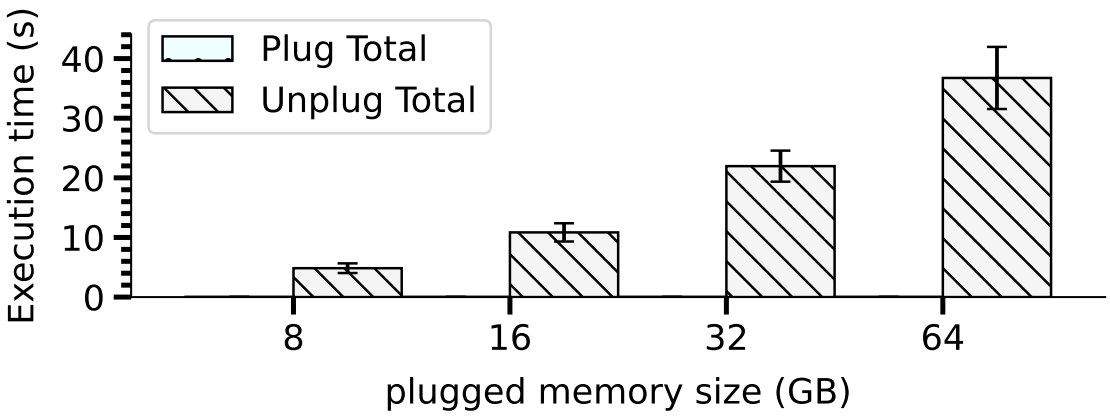
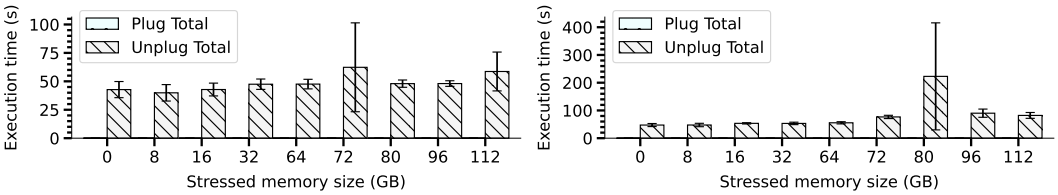


Figure 7.7: Hot-add/Hot-remove delays as a function of (un)plugged memory



(a) After stress job

(b) Concurrent stress job

Figure 7.8: Hot-add/Hot-remove delays as a function of (un)plugged memory

16 vCPUS VM and run a stress job with different memory stress size ranging from 0 to 112 GiB. For stress sizes below 64 GiB, pages allocation may in theory be issued exclusively on boottime memory. For stress sizes above 64 GiB, pages allocation will be performed on hotplugged memory. We can see that unplug time is constant in average for the different memory stress size.

7.1.3.4 Detailed analysis of delays

We try to determine the underlying time contributions responsible for long delays especially for memory hotunplug which is much longer than memory hotplug.

Description. In this experiment, we start a 16 vCPUs VM with 64 GiB memory and 64 GiB hotpluggable virtio-mem memory automatically onlined to ZONE_NORMAL. The VM thus starts with 128 GiB memory and we hot-unplug the 64 GiB of memory and monitor the time contribution of the different mechanisms. In this profiling phase, we do not run any stress job. The details of the experiments are presented in chapter 8.3.2.1. A summary of these results is presented in Table 7.2.

	hot-plug	hot-unplug
Hypervisor	Negligible	
madvise		3 %
EPT fault handling		95 %
→ 4k page-zeroing		60%
→ Huge Page zeroing		18%
→ PUD splitting		2 %
Hyperv. Total (number of CPU cycles)	Negligible	12,610,250,000
Guest		
Guest-to-host request (SBM unplug)		0.36 %
Memory block removal and offlining		4.2 %
→ Remove pages of buddy		0.3 %
→ Offline device pages		3.9 %
Defragmentation (Isolate pages)		93%
→ per-page kmap() in kernel		4 %
→ page-zeroing		89%
Guest Total (number of CPU cycles)	511,500,000	67,010,118,897
Total (number of CPU cycles)	511,500,000	79,620,368,897

Table 7.2: Overall time contributions for virtio-mem

In this section we have reviewed ballooning, free-page-reporting and memory hotplug, the most common mechanisms used in virtualized environments to dynamically change available memory resources available to the guest. We propose a study of configuration change speed for each mechanism, and we profile the cost of each mechanism to examine the cause of the expensive response time.

In our study of these mechanisms, we have identified two sets of mechanisms. The first set is ***hypervisor driven*** *mechanisms i.e. mechanisms that perform configuration changes after receiving a request from the hypervisor such as ballooning and memory hotplug. The second set is* ***guest driven*** *mechanisms i.e. mechanisms that let the guest issue proactive actions to reduce its memory footprint like free-page-reporting. We have discussed the limits to hypervisor mechanisms which only provide the raw mechanisms without providing the control logic. The control logic typically requires* ***feedback control*** *algorithms which trades response time for higher hysteresis and increased CPU overhead.*

7.2 Hypervisor tiering semantic gap

In previous section we have reviewed response time and speed limits to dynamic memory changes to virtual machines. Such changes are required to best adapt virtual machine memory to their actual consumption. We have shown how hypervisor decision-making introduces a trade-off between CPU consumption and response time.

In this section, we study how hypervisor-level memory management leads to sub-optimal decision-making because of information loss. In particular, we try to assess the cost of memory tiering at hypervisor level.

7.2.1 Uncooperative hypervisor swapping

As reviewed in section 2.4, swapping is an historical method used to offload memory pages to a slower storage backend. It has been recently used to perform accesses to remote memory in various work [57, 61, 7]. Since swap-based systems have been used to let processes and containers access remote memory, we may wonder how well does remote memory swapping perform with virtual machines. Swapping typically works by registering a free storage device partition or a file against an operating system. In virtual machines, a swap device may be registered at hypervisor level or guest level.
At hypervisor level, a swap device enables to grow the memory capacity available to support page allocation for all virtual machines. Hypervisor level swapping is convenient to support overcommitment i.e. running a total VM memory size larger than available host memory without causing VM crashes.
On the other hand, guest level swapping enables to grow the memory capacity of the virtual machine. Thus, it would rather be used in a scenario where the guest is responsible for overusing memory and offloading it to its own storage backend.

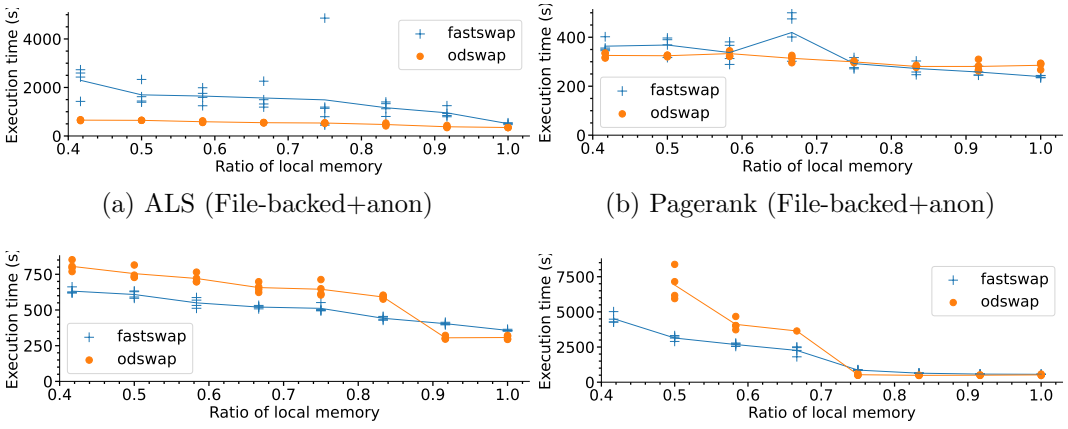
In Figure 7.9, we try to assess if using existing remote memory swap prototype using RDMA like fastswap [7] at hypervisor level to access remote memory is suitable for remote memory accesses. In particular, we try to determine if issuing hypervisor page offloading degrade performance compared to guest level offloading.
In this experiment, we compare fastswap and our own guest-level swap prototype (ODswap). We try to determine how these two prototypes perform for a set of four applications:
We are interested in getting the local memory ratio which stands for the amount of memory used locally compared to the overall memory required to execute the program (known as resident set size). We can configure the local memory ratio using memory cgroups to constraint the memory available to the process and force the swapper to evict pages.

In Figure 7.9, we can see that for ALS, an IO intensive application which reads a dataset. It goes from around 349s for 100% local memory ratio for rmem and fastswap to 644s with rmem and 1695s with fastswap for 50% local memory ratio.
For kmeans, when 100% of memory is local, execution time is 561s, while for 50% local memory ratio with rmem execution time is 6911 s and 3142 s for fastswap.
For pagerank, when 100% of memory is local, execution time is 285 s, while for 50% local memory ratio with rmem execution time is 324 s and 368 s for fastswap.

Application	#threads	AppRSS	Description
quickort	1	8 GiB	C++ quicksort on integers
k-means	16	8 GiB	Python scikit [117] k-means
pagerank	16	8.6 GiB	Spark [168] GraphX pagerank on wikipedia categories subgraph
Alternating Least Square (ALS)	16	21.2 GiB	Spark [168] ML recommendation algorithm on MovieLens dataset

For quicksort, when 100% of memory is local, execution time is 357 s, while for 50% local memory ratio with rmem execution time is 755 s and 608 s for fastswap.

This experiment illustrates the page metadata information loss between hypervisor and guest OS. While guest OS is aware of mapping type (file-backed or anonymous), the hypervisor has lost this information. But, file-backed pages and anonymous pages require separate logics for offloading. Indeed, a clean file-backed page may be either dropped because it has already been persisted on a storage backend or offloaded for faster access than on storage backend. Moreover, a dirty file-backed page may be written back on storage backend instead of being offloaded to tiered memory. On the contrary, dirty and clean anonymous pages need to be offloaded, but they may never be dropped without causing process crash.



(a) ALS (File-backed+anon)

(b) Pagerank (File-backed+anon)

(c) Quicksort (Anon)

(d) Kmeans (Anon)

Figure 7.9: ODswap vs. fastswap

7.2.2 Uncooperative page tiering

In chapter 1, we have presented the recent progress in cache coherent interconnects such as CXL [34, 60] (see §1.5.3). It has been reported in various papers [60, 34, 95, 104] that CXL memory accesses share similar semantics to NUMA memory accesses. Indeed, it is expected from an OS perspective that CXL memory accesses will be cachable MMIO-like accesses. Some papers such as TPP [104] or Pond [95] even rely on simulation of a CXL.mem device as a CPU-less NUMA node. Thus, we

Type	Exposed RR	Exposed FT	Unexposed FT	Unexposed RR
vNUMA nodes	4	4	1	1
vCPUs	64	64	64	64
host memory (GiB)	64	64	64	64
NUMA memory policy	strict binding	strict binding	first-touch	interleaved

Table 7.3: Summary of VM configurations

believe NUMA offers relevant similarities to observe page placement consequences of future rack scale interconnects.

Moreover, in section 3.5 we have reviewed how OS needs to map uniform virtual pages on heterogeneous physical pages by performing automatic page placement. However, running VMs on NUMA machines adds a second level of complexity to memory management since both guest OS and hypervisor needs to issue page allocations. It is common for NUMA-aware hypervisors to rely on vNUMA, an emulation of the host NUMA topology exposed to the guest. In this section, we try to assess how topology information is beneficial for the guest to perform optimal placement.

There exist two main *static NUMA policy* available for page allocation (mostly during page fault handling) at OS level which are *first-touch* and *Round-Robin*. *First-touch* (FT) policy allocates a physical page on the same NUMA node as the CPU issuing the request.
Round-Robin (RR) policy tries to allocate physical pages in a round-robin way on the set of NUMA nodes. There exists also dynamic policies commonly implemented as userspace daemons trying

7.2.2.1 The influence of exposed vNUMA topology

In Figure 7.10 we measure the speedup of applications in a VM configured with different NUMA settings for a set of applications from the Parsec [22] and NAS parallel benchmark [18] benchmarking suite over 30 iterations. The VM is restarted between each run to prevent hypervisor page allocation pollution across runs.
The host machine is an Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz with 4 NUMA nodes each made of 16 cores per NUMA node with 2 hyperthreads per core. Each NUMA node has 64 GiB of memory on each NUMA node. All VMs are started with 64 CPUs and 64 GiB of memory, and we configure them according to Table 7.3. NUMA balancing and NUMA migration are disabled for this experiment.

In Exposed configuration, vCPUs are allocated on hypervisor threads pinned on the same NUMA node as the vNUMA node DIMM. vDIMMs are allocated on an hypervisor NUMA node and a strict policy is used to prevent out-of-node allocations.

In Unexposed configuration, vCPUs are allocated on hypervisor threads pinned on the different NUMA nodes proportionally with 16 hypervisor threads per NUMA nodes. vDIMMs are allocated uniformly with first-touch policy or interleaved policy across hypervisor NUMA nodes.

Figure 7.10 reports application speedup for the different configurations. The

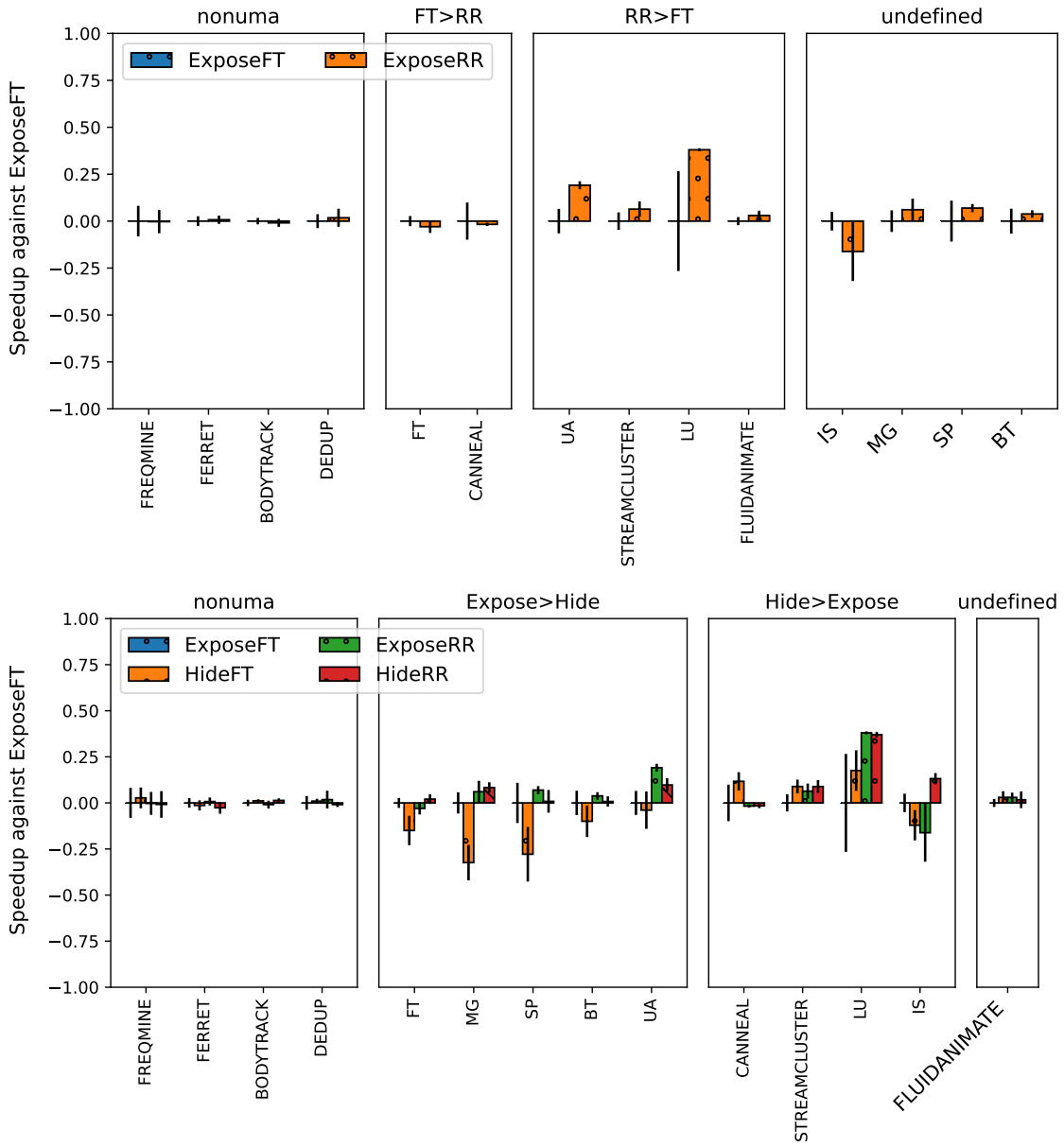


Figure 7.10: Performances of applications for Round-Robin vs First-Touch NUMA policies and Exposed and Unexposed NUMA topologies

graphic gives an overview of our study but remains impractical for interpretation thus we rely on statistical tooling to exhibit significant differences.

7.2.2.1.1 Classifying results

Since, it is hard to visually classify if a NUMA configurations is significantly quicker than another configuration, we rely on a statistical testing method to determine if two distributions are significantly different. In our method, we use *one-way Welch analysis of variance (ANOVA)* [160] for the *execution time* as a dependent variable and *NUMA configuration* as an independent variable. We then perform Games-Howell test [55] to examine the difference of means between the two distri-

butions.

One-way ANOVA requires that the distributions studied are *normal distributions* which validate visually for each plot. It also requires that samples are *independent* which is validated by restarting the VM between each measurements. Each group has an equal sample size of 30 measurements. However, population variances are different which violates the homoscedascity assumption of ANOVA. Thus we rely *Welch ANOVA* and on *Games-Howell* tests which do not require homoscedascity assumption.

7.2.2.1.2 Effect of FT vs RR policies

	$\overline{Exp.FT}$	$\overline{Exp.RR}$	diff	se	T	df	pval	hedges
BT	35.74	34.38	1.36	0.45	3.02	34.19	0.00	0.77
CG	15.95	15.10	0.85	0.14	6.23	43.93	0.00	1.59
FLUIDANIMATE	61.22	59.39	1.83	0.37	4.88	56.31	0.00	1.24
FT	9.26	9.53	-0.28	0.07	-3.81	55.76	0.00	-0.97
IS	0.50	0.58	-0.08	0.02	-5.36	34.82	0.00	-1.37
LU	42.40	26.28	16.12	2.06	7.83	29.05	0.00	2.00
MG	6.14	5.77	0.37	0.09	4.03	57.97	0.00	1.03
SP	42.30	39.36	2.94	0.86	3.43	31.29	0.00	0.87
STREAMCLUSTER	126.74	118.64	8.10	1.43	5.65	57.10	0.00	1.44
UA	41.48	33.55	7.93	0.52	15.28	35.05	0.00	3.89
BODYTRACK	88.97	89.79	-0.81	0.44	-1.83	55.51	0.07	-0.47
CANNEAL	68.70	69.88	-1.18	1.25	-0.95	29.44	0.35	-0.24
DEDUP	30.77	30.22	0.55	0.34	1.61	54.48	0.11	0.41
EP	2.73	2.72	0.01	0.03	0.46	57.98	0.65	0.12
FERRET	27.04	26.84	0.20	0.17	1.18	56.58	0.24	0.30
FREQMINE	31.23	31.31	-0.08	0.58	-0.15	54.13	0.89	-0.04

Table 7.4: Games-Howell test for Expose-FT and Expose-RR

Table 7.4 reports in first half all the applications which have significant differences between ExposeFT and ExposeRR and in the second half of the table the applications with no differences. $\overline{ExposeFT}$, $\overline{ExposeRR}$ reports the average of execution time of various sample for these two groups. $diff$ reports the difference between the two averages. *se* is the standard error for the distribution of the difference of both groups. T is the T value which stands for the difference divided by the standard error. *df* the adjusted degree of freedom. *pval* the Games-Howell p-value with p-value under 0.05 showing significant differences between distributions. We can observe that BT, CG, FLUIDANIMATE, FT, IS, LU, MG, SP, STREAMCLUSTER, UA are all reported to have significantly difference between expose RR and expose FT configurations, but the other workloads like BODYTRACK, CANNEAL, DEDUP, EP, FERRET, FREQMINE are not statistically different for RR and FT policies. This shows that **some workloads will not benefit from performance improvements out of these policies**.

We can also observe by looking at the sign of hedges that some workloads are more efficient using round-robin policy such as BT, CG, FLUIDAMINATE, LU, MG, SP, STREAMCLUSTER, UA. On the other hand, some workloads are more efficient using first-touch policy such as FT and IS. This shows that **there is no**

	$\overline{Exp.FT}$	$\overline{Unexp.FT}$	diff	se	T	df	pval	hedges
BODYTRACK	88.97	87.99	0.98	0.31	3.13	43.33	0.00	0.80
BT	35.74	39.30	-3.56	0.71	-5.05	54.56	0.00	-1.29
CANNEAL	68.70	60.65	8.05	1.39	5.80	42.75	0.00	1.48
CG	15.95	14.64	1.31	0.16	8.35	56.27	0.00	2.13
EP	2.73	2.58	0.15	0.02	6.76	38.16	0.00	1.72
FLUIDANIMATE	61.22	59.39	1.83	0.45	4.12	49.51	0.00	1.05
FT	9.26	10.64	-1.38	0.14	-9.74	35.56	0.00	-2.48
IS	0.50	0.56	-0.06	0.01	-6.89	47.92	0.00	-1.76
LU	42.40	34.95	7.45	2.23	3.34	38.67	0.00	0.85
MG	6.14	8.13	-1.99	0.13	-15.79	47.47	0.00	-4.03
SP	42.30	54.08	-11.78	1.42	-8.29	53.23	0.00	-2.11
STREAMCLUSTER	126.74	115.46	11.28	1.38	8.18	55.40	0.00	2.08
DEDUP	30.77	30.46	0.30	0.22	1.37	35.88	0.18	0.35
FERRET	27.04	27.41	-0.37	0.19	-1.97	57.71	0.05	-0.50
FREQMINE	31.23	30.37	0.85	0.56	1.51	51.64	0.14	0.39
UA	41.48	43.10	-1.62	0.91	-1.78	49.49	0.08	-0.45

Table 7.5: Games-Howell test for Expose-FT and Unexposed-FT

rule-of-thumb policy and workloads may best behave under RR or FT policy.

7.2.2.1.3 Effect of exposing NUMA topology as vNUMA

Table 7.5 reports in first half all the applications which have significant differences between ExposeFT and UnexposedFT and in the second half of the table the applications with no differences.

First, we can see that BODYTRACK, CANNEAL, CG, EP, FLUIDAMINATE, IS, LU have statistically longer execution time for ExposeFT configuration compared to UnexposedFT. All these applications except IS perform better under RR policy or they have been shown to not perform better between RR and FT policies. Second, we can see that BT, FT, MG, SP have statistically longer execution time for UnexposedFT. Third, some applications such as DEDUP, FERRET, FREQMINE, UA are not impacted by the exposition of the NUMA topology. This shows that **exposing the topology can lead to better execution time if the correct policy in the guest (FT or RR) is selected otherwise performance can be even worse.**

In Table 7.6, First, CANNEAL, FLUIDAMINATE, FREQMINE and MG workloads do not show statistical differences between ExposeRR and UnexposedRR configurations. Second, BT, DEDUP, FERRET, LU, SP, UA have longer execution time for UnexposedRR configuration compared to ExposeRR. However, BODYTRACK, CG, EP, FT, IS, STREAMCLUSTER have longer execution time for ExposedRR over UnexposedRR and FT and IS workloads are supposed to behave well under RR policy.

	$\overline{Exp.RR}$	$\overline{Unexp.RR}$	diff	se	T	df	pval	hedges
BODYTRACK	89.79	87.76	2.02	0.39	5.17	43.94	0.00	1.32
BT	34.38	35.46	-1.08	0.22	-4.87	52.51	0.00	-1.24
CG	15.10	14.83	0.27	0.07	3.69	47.80	0.00	0.94
DEDUP	30.22	31.04	-0.82	0.28	-2.94	33.31	0.01	-0.75
EP	2.72	2.59	0.13	0.02	5.70	40.51	0.00	1.45
FERRET	26.84	27.75	-0.91	0.19	-4.76	51.42	0.00	-1.21
FT	9.53	9.07	0.47	0.07	6.52	55.21	0.00	1.66
IS	0.58	0.43	0.15	0.01	10.04	31.11	0.00	2.56
LU	26.28	26.71	-0.42	0.13	-3.17	43.67	0.00	-0.81
SP	39.36	41.95	-2.59	0.50	-5.13	36.09	0.00	-1.31
STREAMCLUSTER	118.64	115.49	3.15	1.25	2.52	56.77	0.01	0.64
UA	33.55	37.44	-3.89	0.32	-12.12	46.38	0.00	-3.09
CANNEAL	69.88	69.80	0.08	0.19	0.44	52.43	0.66	0.11
FLUIDANIMATE	59.39	60.18	-0.78	0.58	-1.35	45.99	0.18	-0.34
FREQMINE	31.31	31.48	-0.17	0.54	-0.32	56.65	0.75	-0.08
MG	5.77	5.63	0.14	0.07	1.91	41.84	0.06	0.49

Table 7.6: Games-Howell test for Expose-RR and Unexposed-RR

In this study, we have performed a comparison of execution time in various workloads under different vNUMA configuration with topology exposed or unexposed and first-touch or interleaved policy. We have observed that:

1. *Some applications perform identically in first-touch or round-robin policy;*
2. *Some workloads are more efficient using round-robin while others are better using first-touch;*
3. *Exposing the hypervisor topology to the guest using vNUMA generally yield performance gains (between 0 to 30%) if the appropriate application policy is selected in the guest but exposition of the topology with incorrect policy can also lead to worse performances than hypervisor-level NUMA policy.*

This study has been conducted using coarse-grained static policy (first-touch and round-robin) for page allocation. However, such static policy may cause suboptimal placing and can have high variability across runs, and it is expected that per-application fine-grained tuning may offer better performances.

In this chapter, we have provided an overview of virtual machines techniques which can be used to dynamically adapt virtual machines memory to actual guest usage. We have performed a top-down analysis of time contribution of existing mechanisms. We have shown how existing mechanisms introduce long delays to perform simple memory add and remove operations. We have reviewed guest-driven and hypervisor-driven approaches to reduce the memory footprint of a virtual machine and discussed their limits. In particular, we have exhibited that hypervisor-driven approaches needed to rely on feedback control algorithms with continuous probing of guest statistics and with information loss. We have shown that this approach could introduce further reactivity delays undesirable for dynamic memory changes. On the contrary, guest-driven approaches fails to shrink its caches automatically by lack of knowledge of available host memory available.

In a second part, we have conducted experiments to try to assess the cost of information loss between guest OS and hypervisor in tiered memory management. We have studied swap-like remote memory solutions at hypervisor and guest level to show that hypervisor-level memory management can misbehave when page mapping information is lost. We have also studied NUMA-like remote memory solutions to show how topology information as well as application preferred policy for heterogeneity could yield better execution time.

As a conclusion, this chapter has identified multiple limits to the dual memory management of virtual machines. Based on these limitations, chapter 8 discusses our contributions to try to tackle these issues.

8 ExoVM, fast elastic VMs

In chapter 7, we have motivated the co-design of VM and hypervisor with a review of two major limitations with current VMs to improve resource usage.

The first limitation prevents VM to quickly reconfigure, during execution, the amount of resources they can use. We have seen that this limit is caused by mechanisms which are too slow to shrink or grow VM memory compared to the speed of a VM to consume memory. We have detailed the subsystems responsible for slowing down memory reconfiguration.

The second limitation describes performance degradation of applications running in virtual machines which try to use far memory to increase memory usage in the rack. We have seen that performance degradation is caused by the existence of two layers of memory management in the guest and the host which perform independent uncollaborative memory management decisions. In particular, in §7.2.2, we have evaluated the cost of uncollaborative memory management scenario for swap-based disaggregated memory prototypes (as presented in §4.3.1) while in §7.2.2 we have evaluated problems for prototypes using a cache coherent interconnect (e.g. NUMA, CXL).

In this section, we present ExoVM, a work in progress to support reactive reconfiguration of memory capacity in VMs and finer control over page placement on memory tiers. We first propose to review the design of ExoVM, before discussing the implementation and finally we present the impact of ExoVM in various applications.

8.1 Design

Our discussion so far has illustrated the problem of memory waste in virtual environments but the causes and solutions are very diverse. We have seen that memory waste is mostly caused by memory resource being statically provisioned in each VM at boot time. We have seen that static allocation was amplified by overallocation of memory from users to avoid crashes caused by memory spikes.

As illustrated in chapter 7, there are two main limits to the widespread use of dynamic VM instances which are ***estimating desired guest memory*** *and* ***slow adaptation mechanisms****. Thus, ExoVM addresses the delay of desired memory estimation and proposes to directly let guest applications issue provisioning request*

which saves many CPU cycles in feedback control. ExoVM also reduces the delay of memory hot-plug and hot-unplug by skipping the addition of pages to the page allocator.

In this section, we first propose a simple model to understand the reasons behind static usage of memory resources and how memory management safety is trivialized by the use of static VMs but how it can be more challenging with techniques supporting dynamic reconfiguration at execution time (e.g. memory ballooning). Second, we discuss performance consideration in the design of dynamic memory usage by leveraging arguments presented in chapter 7. Third, we present a general overview of ExoVM to support safe and efficient execution with guest-initiated memory allocation and freeing which are historically delegated to the hypervisor. Fourth, we discuss the design of memory resource revocation for collaborative and uncollaborative guests inspired by exokernel approach. Fifth, we justify the use of 1:1 mappings between process segments and physical memory slots. Sixth, we discuss the advantages of our approach to support inter-VM shared memory segments.

8.1.1 Safety in dynamic memory management

We propose a short model of VM memory to help capture the challenges in safely balancing memory across VM according to their needs over time. In our model, each VM is described with with two parameters: **memory capacity** and **memory usage**.

8.1.1.1 Model.

First, each VM is bound to a **memory capacity** ($Capacity_{VM}(t)$) which defines the total amount of physical pages available in the VM to satisfy page allocation requests. Each VM is started with an initial capacity ($Capacity_{VM}(t0)$) which corresponds to the VM allocation request issued by the VM scheduler. As illustrated with memory ballooning and memory hotplug (see section 7.1), memory capacity can be changed during VM execution following an hypervisor request. Notably, in section 7.1, we have discussed the particularities of each technique to adapt VM capacity: *Legacy memory hotplug* provides a straightforward approach to memory capacity changes with the hypervisor adding or removing DIMM made visible to the guest through firmware topology (SRAT) and notification (ACPI). *Paravirtual memory hotplug* also rely on the hypervisor to add/remove DIMMs, but it introduces the hypothesis of *guest kernel collaboration* to exclude or include sub-DIMMs from the guest memory management to support finer-grained adaptation of *Capacity*. *Memory ballooning* fully relies on the assumption of a *guest kernel collaboration* by letting the hypervisor communicate a request to the guest to change its memory capacity.

Second, each VM **consumes physical memory** during its execution ($Used_{VM}(t)$) which occurs as a result of an allocation request to the *page allocator*. Conversely, guest physical memory may be released by issuing a page freeing to the *page allocator*. Interestingly, because of the *duplication of memory management*, when a page is used in the guest and later freed, it is not freed in the hypervisor. In §7.1.2, we have described how free-page-reporting implements a notification mechanism

integrated to page freeing to enforce that each time $Used_{VM}(t)$, hypervisor used memory $Used_{Host}(t)$ is also decremented. In section 5.5, we have reviewed various studies pointing out that memory usage in VMs is unpredictable on the hypervisor side. Existing paravirtual devices (virtio-balloon) supports communicating guest used memory to the hypervisor making $Used_{VM}(t)$ observable from the hypervisor at the expense of a communication delay.

8.1.1.2 Defining safety.

The safety property the hypervisor must enforce is that an hypervisor should not kill a VM.

By definition, VM used memory is always limited by the VM memory capacity (i.e. $Used_{VM}(t) < Capacity_{VM}(t)$). When used memory grows close to the memory capacity, the guest will invoke guest-level memory reclamation mechanisms (see section 2.4). Similarly, when host used memory grows close to the host capacity, the host invokes hypervisor-level memory reclamation. We have shown in section 7.2 that hypervisor-level memory reclamation is undesirable because of lack of understanding of guest memory usage. Additionally, failure to reclaim pages at hypervisor level result in preemption of a VM with no notification which lets absolutely no chance for guest kernel reclamation or application reclamation.

Based on $Capacity(t)$ and $Used(t)$ memory, we can now clarify the different scenario which may cause safety violation.

First, an immediate safety violation occurs when all VMs running on the machine are using more memory than available in the host ($\sum_{v \in VM} Used_v(t) > Capacity_{host}$). This scenario is hardly solvable at the scale of a single server and lets two solutions. Either, the host capacity is extended (e.g. using memory disaggregation) or a VM is removed from the set of running VMs (e.g. VM migration or killing a VM).

Second, there exist a scenario where VM safety is not immediately violated but may be violated in the future. The conditions which leads to this scenario is when total VMs capacity is higher than host capacity ($\sum_{v \in VM} Capacity_v(t) > Capacity_{host}$) but VMs are using less than the host capacity ($\sum_{v \in VM} Used_v(t) < Capacity_{host}$) which means the host can still run VMs without any crash.

Thus, in order to guarantee execution safety, the hypervisor must maintain at all times the guarantee that VMs capacities do not exceed the host capacity:

$$\sum_{v \in VM} Capacity_v(t) < Capacity_{host} \tag{8.1}$$

8.1.1.3 Safety in classic VM memory management

The classic VM model, which is widely used in datacenters, relies on a large simplification of this problem by statically setting $Capacity_{VM}(t)$ to an initial value. Safety is ensured a single time at allocation time by the VM scheduler which verifies that $\sum_{v \in VM} Capacity_v(t_0) < Capacity_{host}$. However, in this model, when $Used_{VM}(t)$ reaches $Capacity_{VM}(t_0)$, there is no way to increase $Capacity_{VM}(t_0)$ even if remaining memory in the host ($Capacity_{host} - Used_{host}(t)$) would be able to satisfy the allocation request.

8.1.1.4 Safety in VM memory overcommitment architecture

In existing VM memory overcommitment mechanisms, such as memory hotplug or memory ballooning, each user defines a minimum and a maximum for $Capacity_{VM}(t)$: $CapacityMin_{VM}$, $CapacityMax_{VM}$. The hypervisor maintains during the execution of each VM: $Capacity_{VM}(t) \in [CapacityMin_{VM}, CapacityMax_{VM}]$.

Bounding VM capacity between a minimum and a maximum gives more flexibility to support safe memory overcommitment:

$$\sum_{v \in VM} Capacity_v(t) < Capacity_{host} \tag{8.2}$$

$$\sum_{v \in VM} CapacityMax_v > Capacity_{host} \tag{8.3}$$

$$\forall v \in VM, Capacity_{VM} > CapacityMin_v \tag{8.4}$$

8.1.1.5 Safety in ExoVM

In ExoVM, we guarantee the safety property $\sum_{v \in VM} Capacity_v(t) < Capacity_{host}$ at all time by using a memory server which maintains accounting on each VM capacity.

Capacity of a VM may be decremented either by a collaborative guest or from the hypervisor. If a collaborative guest decrements capacity, it issues unmapping of the process segment and unplugging and freeing of the memory region. However, since guests may be uncollaborative, the hypervisor may notify guests to perform revocation or directly call abort protocol. The details of the abort protocols are still work in progress.

In this section, we have proposed a basic model and reviewed the different architectures to ensure safe execution while maintaining performances. In the next section, we define how to avoid trivial proposals of safety which try to minimize guest capacity by detailing the performance conditions.

8.1.2 Performance condition: Desired memory

The safety condition ensures that no VM will be killed by the hypervisor and that only the guest triggers memory reclamation. However, this condition alone does not define performance objectives to best use local memory. Indeed, it would be possible to set low $Capacity_{VM}(t)$ objectives in all VMs to ensure safety, however, this would result execution time in all VMs being spent in guest memory reclamation. Moreover, $Used_{VM}(t)$ is guaranteed to be less than $Capacity_{VM}(t)$ by construction of guest memory management.

In order to provide optimal performance execution, a VM needs to determine a desired memory usage. Desired memory corresponds to the future used memory in the guest if the VM was living under infinite capacity. There exists different approach to estimate desired memory which we discuss in the next paragraphs.

8.1.2.1 The problem of inferring desired memory with hypervisor management

In section 7.1, we have detailed the culprit for the delays to enforce $Capacity_{VM}(t)$ changes in existing mechanisms (i.e. ballooning and hotplug). We have seen that these delays were caused by mechanisms themselves and feedback control running in the hypervisor which tries to automate the use of these mechanisms. In the next paragraph, we dwell on the cost of automation through feedback control.

In §7.1.1.4, we have seen that hypervisor memory overcommitment mechanisms rely on feedback control algorithms to adapt $Capacity_{VM}(t)$ in each VM. We have shown that feedback control contributes an additional time overhead to raw mechanisms which further degrades responsiveness for elasticity.

The time contribution of feedback algorithm is partially caused by trying to infer at hypervisor level what is the desired memory capacity in the next time units. Feedback control tries to determine if $Used_{VM}(t)$ is growing or decreasing over the last time window prior to issue changes to $Capacity_{VM}(t)$. Indeed, guest applications are knowledgeable of how much memory they want to use in the future by performing memory allocations which are converted to process segment creation (mmap()) or extension (brk()) after going through memory allocation libraries. These operations on process segments are directly satisfied by the guest kernel through system calls but they are never propagated to the hypervisor. Then, the hypervisor needs to infer the future memory usage by monitoring $Used_{VM}(t)$ over multiple time units.

Feedback control overhead can not be entirely explained by the inference of future desired memory. Additional overheads can be explained misprediction of future memory usage, communications overhead caused by monitoring guest memory statistics, the implementation of the feedback control algorithm which tries to determine the new $Capacity_{VM}(t)$ that needs to be injected in the VM.

8.1.2.2 ExoVM exokernel approach with guest-initiated capacity changes

Based on the observation that long delays to change $Capacity_{VM}(t)$ would result in more frequent unsafe scenario, we try to remove the delays introduced by feedback control. In ExoVM, we propose to let VMs directly extend and revoke their physical resources. Our intuition proposes a variation of the exokernel [49, 79, 56] OS architecture which proposes to let processes directly access physical resources by manipulating secure bindings on hardware resources.

In ExoVM, we propose to instrument process segment creation and extension by intercepting system calls on process segments. Indeed, virtual memory management requires creating a segment prior to performing a memory access. We translate intercepted calls to allocations or freeing of ranges of the physical address space to avoid the time-contribution of feedback-control algorithm. Thus, when a VM requires more memory, we increment VM capacity by the size of the process segment.

8.1.3 ExoVM, fast and dynamic memory management

In Figure 8.1, we illustrate the main design ideas of ExoVM. Figure 8.1 first shows that ExoVM maintains backward compatibility for processes (e.g. P1) so that they

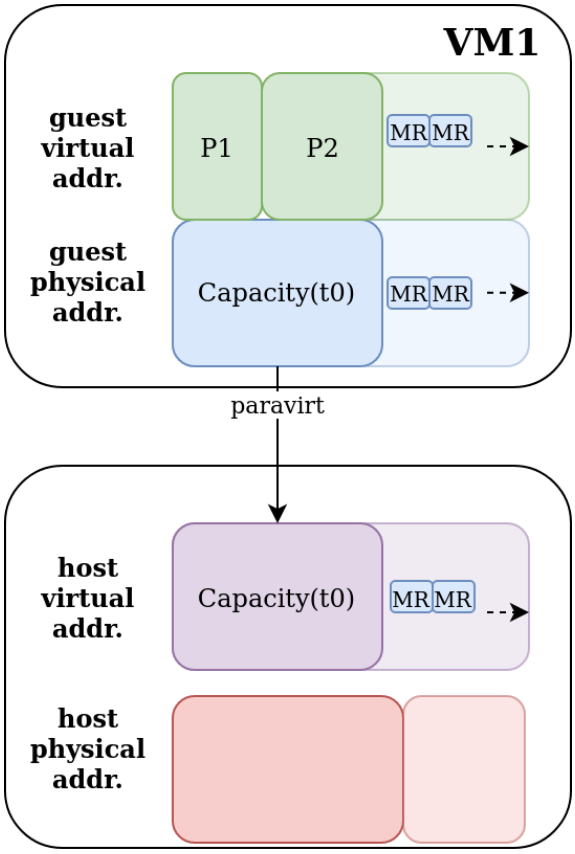


Figure 8.1: Overview of ExoVM design

can still use anonymous and file-backed memory. It shows that VM are instantiated with an initial capacity ($Capacity(t_0)$) which can be dynamically modified over time to extend or shrink the guest physical address space. Communications between the guest and the host are delivered over paravirtualization queues as already proposed in ODswap. Since VMs run in processes, extension or shrinking in the guest physical address space require an identical operation on the host virtual address space. The final consumption of physical memory at hypervisor level is at most the current capacity $Capacity(t)$ of the VM and is usually lower because of on-demand paging.

8.1.4 The problem of memory resources revocation

Adding new memory resources to a VM is made very easy with the use of our new interface. However, it also makes it faster to reach the host memory limits by continuously adding more memory. When using more memory comes with no constraints, there is no reasons for a guest to free memory capacity. Moreover, some guests may be compromised and may try to willingly acquire large memory resources to starve other collocated VMs. Thus, ExoVM requires a solution to incite guests to free memory and ultimately to reclaim the resource.

Freeing memory could be encouraged by applying fares on memory usage over time, however, this approach is unlikely to be satisfactory as memory leaks or bugs could have expensive consequences. Thus, a more realistic approach can be im-

plemented using the *revocation* and *abort* semantic proposed in exokernels. As a summary, exokernels propose *visible revocation* to notify collaborative processes to release resources collaboratively. Since, processes may be uncollaborative, they may not revoke resources, in which case, the exokernel is supposed to initiate the *abort* protocol which breaks the grant a process has acquired.

In our case, it would be unreasonable to assume full collaboration from guest applications, however raising awareness about capacity pressure would enable fine-grained decision making at application level. Concrete implementation of visible revocation and abort is already proposed in resource deflation [132].

8.1.5 Bypassing guest page allocator: 1:1 mapping of process segments on physical memory slots

Legacy memory hotplug and paravirtual memory hotplug add plugged memory to Linux memory zones managed by the buddy allocator. The pool of plugged guest physical pages later serves to satisfy page allocation request from any user-space guest process and may also be used to satisfy in-kernel guest memory allocations (if pages are added to NORMAL zone). In this section, we present the reasons behind ExoVM proposals to enforce 1:1 mappings between process segments and physical memory slots.

8.1.5.1 Flattening memory management

Guest page allocator duplicates many services which already exists at hypervisor level. In particular, page allocation is provided by hypervisor-level buddy allocator. We have also discussed the problem around guest-only page freeing which results in increasing memory usage from hypervisor perspective. Free-page-reporting implements hypervisor-level page freeing by buffering page freeing operations. Thus, ExoVM maps process segments on physical memory slots to bypass the guest page allocator.

8.1.5.2 Easier memory revocation

Memory revocation is initiated by the hypervisor to call guest application logic to try to free memory pages with application memory layout knowledge. Memory revocation without 1:1 mapping would be hard. Indeed, without 1:1 mapping, a memory region could back pages in different processes and conversely a process segment could be backed by multiple memory regions. Thus, trying to remove a memory region would require isolating the subset of pages used by each process and issuing page freeing on each subset. In ExoVM, we use 1:1 mapping of process segments to physical memory slots to make revocation easier.

Revocation of private memory regions. Private memory regions are owned by a single process. Revocation is simply performed by calling the revocation call of the memory region in process context.

Revocation of shared memory regions. When memory regions are shared between multiple processes or virtual machines You can call revocation handler bound to the memory region in the guest application.

8.1.5.3 Custom memory placement and freeing at process segments granularity

There is no real rules regarding similarity of objects in a same process segments. The only guarantee provided by mmap is that all pages backing the objects are either anonymous or file-backed and private or shared. However, there is no assumption about objects properties in a process segment. In details, process segment do not guarantee that allocated objects share **similar lifetime** (as in generational garbage collector). Moreover, they do not guarantee that objects have **similar access frequency**.

ExoVM mostly proposes to build applications memory management to try to fit on a memory region objects which may shared similar lifetime and similar access frequency. Ideally, a developer using ExoVM will use his knowledge of objects access frequency and lifetime to try to ensure that objects on a memory region will have similar properties. However, predicting these properties for each object is not practically doable and thus, ExoVM proposes registration of custom policies to free objects or migrate them on appropriate locations.

8.1.5.4 Bypassing guest page allocator in memory hotplug

Existing solutions for memory hotplug divide plug and unplug operations in two phases. A first phase is dedicated to updating firmware knowledge of available DIMMs and creating/destroy page structures in the kernel for memory management to work. A second phase is dedicated to adding/removing memory pages to the guest page allocator.

The use of 1:1 mappings which bypasses the guest page allocator enables to bypass the first phase. We have seen that in hot-add path, adding pages to the guest page allocator accounts for 98% of the total guest contributions of memory hotplug (in virtio-mem). Even if memory hotplug is already a fast operation, bypass of guest page allocator enables to save 400 ms.

In the hot-remove path, the removal of pages (offline _pages) from the page allocator only accounts for 3% of the total memory hot-unplug time. This represents 594 ms when page allocator is unused for 64 GiB removal. When we run an artificial memory stress job concurrently to a memory hot-unplug operation, we can observe that the time spent shrinking the page allocator can take up to 3000 ms, i.e. 5 times longer than when the page allocator is unstressed. This means that delays to grow and shrink the page allocator capacity are highly impacted by concurrent stress on the page allocator from other processes.

We have observed smaller time contributions for smaller memory changes. However, there is an incompressible cost of around 30 ms which remains even for the smallest memory (128 MiB) changes in online/offline operations. This overhead is unaffordable for workloads which require very frequent dynamic changes as presented as presented in Figure 8.6.

8.1.5.5 Limits to 1:1 mappings

In sections 2.3 and 2.4, we have presented the main types of mappings used in Linux kernel memory management which are file-backed and anonymous mappings. File-backed mappings typically implement 1:1 mappings between file offsets and process segments. However, file-backed mappings do not support direct memory accesses. Both of these mappings rely on kernel memory management mechanisms such as memory reclamation and page allocation through buddy allocator. A 1:1 mappings share part of the semantic of shared file-backed mappings but they are very different from anonymous private mappings commonly used for heap or stack.

Shared mappings and COW. One of the problem in the use of 1:1 mappings is their impact on process forking. Process fork duplicates process information such as virtual memory, opened files, ..., into another process. In order to reduce the overhead of this system call, Linux memory management leverages a technique named copy-on-write (C0W) which performs a lazy copy after each attempt to write to one of the process memory. Copy-on-write is desired as it reduces the memory consumption by sharing identical memory pages between the two processes. Second, COW reduces the latency of fork system call by avoiding full copy of virtual address space at fork time. COW is supported for anonymous private mappings in Linux which we try to replace with 1:1 mappings in ExoVM. Either ExoVM use shared 1:1 mappings, in which case, it will require changes to applications to support synchronization for accesses to the shared segments. Or ExoVM can use private 1:1 mappings, but implementing COW on these mappings will require the allocation of a new memory region by definition which will remove the advantage of COW to reduce memory consumption.

Page zeroing. Converting existing anonymous mappings to 1:1 mapping is also challenging as it breaks the assumption that freshly allocated pages are zeroed. Indeed, anonymous private pages are zeroed in page fault handling when the page is first accessed. A 1:1 mapping assumes that content may have been written on physical memory prior to the creation of the mapping. Thus, pages are not zeroed on 1:1 mappings as there are not zeroed in file-backed mappings.

Breaking page zeroing is problematic as software which build on top of anonymous mappings such as language runtimes or memory allocators may or may not assume page zeroing. For instance, golang runtime does not assume page zeroing and performs zeroing by itself in the runtime. On the contrary, jemalloc [51] rely on the guarantee of pages being zero-filled. These systems rely on madvise with MADV_UNUSED flag to synchronously hand back pages to the OS. And the OS issues zeroing in page fault handling during next read or write access.

8.1.6 inter-VM shared memory

There already exists support for inter-VM shared memory with ivshmem [45]. However, existing solutions requires creation of the shared memory segment at VM initialization. In order to best optimize the usage of memory over time, it would be

interesting to support creation of the shared memory segment and teardown with explicit control from the user.

The infrastructure to support dynamic initialization of inter-VM shared memory segment by a guest can be provided easily by ExoVM memory region. Thus, we integrate an argument to memory region allocations to define if the VM would requires sharing the MR with other VMs.

In this section, we have presented an overview of the main design points of ExoVM. In particular, we have presented ExoVM as a solution to support dynamic capacity changes in VMs while maintaining safety of executions. We have introduced the motivation behind guest-initiated decisions to avoid the expensive cost of inferring guests desired capacity. Then, we present the different components and abstractions proposed in ExoVM to support safe allocations initiated by the guest. Next, we introduce the different motivations behind 1:1 mappings of process segments on physical memory slots. Finally, we proposed to leverage the existing abstractions of ExoVM to support inter-VM memory sharing.

In the next section, we present the details of the implementation of ExoVM.

8.2 Implementation

In section 8.1, we have presented the two software components used in ExoVM to support hotpluggable memory regions.

Similarly to section 6.3, we present the implementation of these two software components. We begin by a presentation of the guest Linux kernel module which exposes a virtio PCIe device to deliver paravirtualized communications with the hypervisor. Then, we present the hypervisor side implemented as a qemu device which implements the server part of the virtio device. Finally, we perform a micro-evaluation study of the individual performances of each memory region operation.

8.2.1 Guest side

As explained before, ExoVM is made of two main communicating components, one in the hypervisor and another one in the guest. In this section, we discuss the implementation of the guest interface between userspace and the kernel.

First, we shortly present ExoVM userspace interface and how it delegates control operations to the kernel. Second, we present how we integrate with the existing DAX system in Linux to support direct memory accesses with hardware MMU to tiered memory. Third, we explain how we support the different operations on guest memory regions with support for allocations, freeing, plugging, unplugging and revocation.

8.2.1.1 ExoVM userspace interface

One of the main interest of ExoVM is to let guest processes directly perform the memory management operations provided by the memory region API. It is also important that guest processes can access the memory resources which are dynamically added to the virtual machine.

In 8.1, we present the userspace interface to ExoVM which proposes userspace hooks to hypercalls.

Listing 8.1: vDIMM API

```
typedef struct vdimm {
        uint64_t gpa;
        int region_id;
} vdimm_t;

int exovm_plug(vdimm_t *vdimm, mr_t *mr);
int exovm_unplug(vdimm_t *vdimm);
```

Listing 8.2: MR API

```
typedef struct mr {
        uint64_t raddr;
        uint64_t rlen;
        uint32_t rkey;
} mr_t;

int exovm_mr_alloc(mr_t *mr, size_t size, uint64_t align,
```

```
                    uint8_t shared);
int exovm_mr_free(mr_t *mr);
int exovm_mr_free_page(mr_t *mr);

typedef void (*reclamation_cb_t)(int);
int rmemctl_register_cb(mr_t *mr, reclamation_cb_t reclamation_cb);
```

Guest-kernel communications. Communications between user-space process and kernel are implemented using netlink, a communication mechanism integrated with BSD sockets. We use netlink since it is inspired by BSD sockets networking primitives to issue message-passing communications between userspace and kernel.

Process segments. In §8.1.5, we discuss ExoVM tries to map process segments directly to physical memory slots. We rely on the existing DAX system in Linux which supports the creation of a special kind of process segment with dedicated page fault management. DAX segments have similar semantics to a shared file mappings however every operation on the virtual address space is directly handled by the MMU without going through software layers like page cache. In the next section, we detail the integration with DAX system in Linux.

8.2.1.2 Implementing 1:1 mapping using DAX

In §8.1.5, we have presented the reasons and challenges to use 1:1 mappings between process segments and physical memory slots in VMs. In ExoVM, 1:1 mappings are implemented using DAX, a component of Linux kernel memory management. We have shorty presented in §3.2.2, the use case for DAX which is mostly used for its capability to bypass page cache by delivering direct memory access on NVDIMMs.

Using DAX. A process can create a DAX segment by calling mmap system call with appropriate DAX flags (i.e. MAP_SYNC and MAP_SHARED_VALIDATE). Linux Kernel appends the mmap segment (struct vm_area_struct) to the process VMA tree and tags the segment as DAX. DAX mappings guarantees that mutations to the DAX segment are directly propagated to the underlying backend without going through software caches (MAP_SYNC).

DAX bus. In Linux, DAX [161] is implemented as a bus abstraction of the Linux Device Driver (LDD) model [38]. In this model, a bus abstraction represents "a channel between the processor and one or more devices". All devices are connected through a bus which can either be physical (PCIe, I2C, USB, SCSI...) or virtual (DAX, workqueues, ...). A bus owns a list of *devices* (struct device) and *drivers* (struct device_driver).

DAX device driver. In Linux device driver model, all system drivers are tracked by Linux to associate them with a device. Thus, DAX provides two implementation of struct dax_device_driver which is a subclass of struct device_driver. The first implementation provides the *devdax* driver which creates a device in Linux devfs

enable process to map this device in their address space as a DAX segment. The second implementation provides the *kmem* driver which supports adding the memory resource capacity of a devdax device to *System RAM*. Transitions from devdax to kmem can be done using daxctl utility. However, the opposite operation requires defragmentation and regularly fails. ExoVM can directly leverage the existing drivers and typically use *devdax* by default.

DAX device. The other components of the bus are devices. DAX devices (struct dax_dev) are a subclass of devices (struct device). Each DAX device owns a physically contiguous range of device memory as well as the infrastructure to support creation of DEVICE zone mappings (pgmap). A DAX device is part of a larger abstraction named DAX region. In ExoVM, we create a new DAX device after each successful plug operation.

DAX regions. DAX regions represent independent devices in the kernel device tree. Regions may be *static* which implies that the user is responsible for assigning a correct numbers (e.g. /dev/daxN.M) for naming of the device and that the device lifetime corresponds to the lifetime of the DAX driver. Dynamic regions supports creation, deletion of DAX regions through sysfs interface (a user-kernel communication interface), and naming of the device relies on allocated numbers. Dynamic regions requires user management of other abstractions such as DAX mappings. ExoVM allocates a new static DAX region each time a new plug operation succeeds. This means that each plug operation results in the creation of a new DAX region and a new DAX device.

8.2.1.3 Support for the different operations

As presented in §8.1.2.2, we directly let guest processes issue allocation, free, plug and unplug requests to the hypervisor instead of trying to infer from the hypervisor when to issue such actions. Allocation, freeing, unplugging and plugging operations are all supported using hypercalls. Revocation of memory region is implemented using an upcall.

The hypervisor sends a response back to the guest containing the results of the allocation request made of guest physical address, length and security key. The response message is received in interrupt context, however registration of the DAX device must be performed out of atomic context. Thus, similarly to ODswap described in section 6.3, we schedule a work for registration of the DAX device.

In ExoVM, we use a single PCIe device to implement paravirtualization connections. The PCIe device is used as a parent node of all children DAX devices created after completion of plug operation. Using a single PCIe device avoids the cost of hot-adding device (acquisition of a read-write semaphore) in the critical path and reduces the memory footprint caused by PCIe device metadata.

8.2.1.3.1 Implementation of MR allocation

When ExoVM receives an allocation request for a memory region, ExoVM performs an allocation hypercall to the hypervisor to request the allocation of memory

in the hypervisor process. At this stage, the memory is not available for the guest and the capacity has not grown.

8.2.1.3.2 Implementation of MR freeing

When ExoVM receives a freeing request for a memory region, it simply forwards the operation to the hypervisor using an hypercall.

8.2.1.3.3 Implementation of MR plug

Each device maintain a list of guest physical address space ranges named **resources** indexed in a **resource tree**. In DAX, the list of resources associated with a device is maintained in the DAX region. In ExoVM, there is a single resource per DAX region and the insertion of new resources is accompanied by the creation of a new device. It is Linux responsibility to insert the device resource in the global resource tree.

Memory is added as device memory and represented by a PCIe device.

8.2.1.3.4 Implementation of MR unplug

When the guest OS receives an unplug request from the process, it first forwards the request to the hypervisor to remove the memory region from qemu memory tree. When the guest OS receives the completion event for the unplug hypercall, it simply implement removal of a memory region in the guest by unregistering the per memory region DAX device, DAX region and IO memory in the resource tree.

Guaranteeing removal of all the kernel structures associated with a memory region is critical to prevent memory leaks. It is particularly challenging to work with static DAX region designed for static resources while ensuring that reference counters on kernel data-structures drop to zero to be freed.

8.2.1.4 Implementation of revocation

ExoVM supports revocation by registering a callback which is registered as a Linux signal with the guest OS. After receiving a notification from an hypervisor upcall, guest OS delivers a signal to the process which triggers the execution of the signal handler in userspace.

This section summarizes the implementation of the memory region interface on the guest side. In the next section, we describe the hypervisor implementation of these operations.

8.2.2 Hypervisor side

In the previous section, we have presented ExoVM interface and their implementation in guest OS. ExoVM communicates with the hypervisor using virtio paravirtualization similarly to what has been described in ODswap. In this section we present the hypervisor side implementation of ExoVM.

8.2.2.1 Hypervisor-side of memory regions

In §8.2.1.3.3, we have discussed the guest side of memory hotplug and in particular, how the guest adapts to expansion and shrinking of its physical address space. In this section, we focus on how the hypervisor-side extends and shrinks guest physical memory. In particular we review qemu abstractions of guest physical memory and how they link with KVM memory abstractions. Then, we present how ExoVM implements plug and unplug calls.

Qemu AddressSpace. Qemu uses an address space abstraction (AddressSpace) to represent a guest physically contiguous memory range from hypervisor knowledge. A VM has multiple address spaces but the two most important are IO and memory address spaces. An address space is made of a tree of memory regions. A memory region describes the result of an hypervisor virtual allocation. It contains various information regarding how the allocation chunk can be used such as support for sharing.

Qemu MemoryRegion. Since memory regions are represented as a tree, there exists two kinds of regions: *container type* and a *leaf type*. Container type is a logical node type in the tree data-structure which is used to store children nodes. Leaf type memory regions can be IO memory regions, RAM, IOMMU ...

ExoVM MemoryRegion sub-tree. We introduce a wrapper around the original qemu MemoryRegion abstraction. In this wrapper, we store state metadata to track whether a MemoryRegion is allocated or plugged.

Additionally, insertion and deletion in qemu memory region tree is not thread-safe. Thus, we isolate a subtree of the MemoryRegion tree on which we synchronize insertion and deletion with a mutex.

8.2.2.2 Support for the different operations

In parallel with the guest support for the different operations presented in §8.2.1.3, this section presents the implementation of the different operations on the hypervisor side.

8.2.2.2.1 Implementation of MR allocation

When the hypervisor receives an allocation requests from the guest, it first allocates the wrapper for the memory region. Then, it uses qemu memory allocator to allocate a memory range in the hypervisor virtual address space as a private anonymous mapping. The allocation request may require a shared allocation, in which case the allocation is simply turned into a shared anonymous allocation in the hypervisor. It appends the memory region to the list of allocated memory region and replies to the guest by providing the address in the host as well as the length and an identifier of the memory region.

8.2.2.2.2 Implementation of MR freeing

It removes the memory region from the list of allocated memory region. Then, it decreases the reference counter on the qemu memory region object which calls the free callback. Finally, it replies back to the hypervisor to inform it of the completion of the operation.

8.2.2.2.3 Implementation of MR plug

When the hypervisor receives a plug request from the guest OS, it first performs an allocation in the guest physical address space to find the next available range. Second, it inserts the memory region in the memory region subtree of the hypervisor which exposes the memory region in the guest physical address space and host virtual address space. Third, it replies to the guest OS with the guest physical address space where the memory region has been installed.

8.2.2.2.4 Implementation of MR unplug

Upon reception of an unplug request from the guest, the hypervisor removes the memory region from qemu memory region subtree. It acquires the lock on the memory region and changes its state to ALLOCATED before appending it into a list of allocated memory regions.

The Memory region finite state machine.

Hypervisor tracks the state for each memory region created from a guest request. Thus, we return errors for the following operations on memory regions: It is forbidden to free a plugged memory region, to free a freed memory region, to plug a plugged memory region (private) and to plug a freed memory region. Similarly, it is forbidden to issue some operations on vDIMMs such as unplugging an unplugged vDIMM, plugging a plugged vDIMM.

This section concludes the implementation of the mechanisms currently supported in ExoVM on the hypervisor side. It has presented how guest requests are treated and how memory management is performed. The next section presents a small guest tool used to perform automatic conversion of memory management in existing applications to use ExoVM.

8.2.3 Supporting the execution of existing applications

ExoVM proposes a new interface for processes memory management to reduce memory usage on a server. Thus, optimal memory management decisions can be achieved but require source code modifications to work with ExoVM. However, it is possible to maintain transparent execution of processes while using ExoVM with simple memory management system calls instrumentation. Thus, we propose to instrument the creation, destruction and expansion of new anonymous memory mappings support the execution of various applications.

Upon creation of a new anonymous segment, ExoVM replaces the segment with the allocation of a memory region of similar size, it then calls plug on the memory region and maps the memory region as a DAX segment in the process. Currently,

no callback is registered with the memory region for revocation of a memory region following an hypervisor upcall. The implementation of a default method is work-in-progress.

Upon teardown of the segment, ExoVM unplugs and frees the memory region.

The support for expansion of an anonymous segment is still work in progress.

8.2.4 Micro-evaluation

This section proposes early micro-evaluation of isolated part of ExoVM to better examine the performances of ExoVM compared to classic VMs.

Description. After presenting the details of the implementation of ExoVM, we try to determine the overhead introduced by **plug** and **unplug** operations on a memory region. In particular, we verify that these operations are not impacted too severely by concurrent operations from threads or processes.

In this experiment, we try to evaluate how plug operation scales with the number of threads.

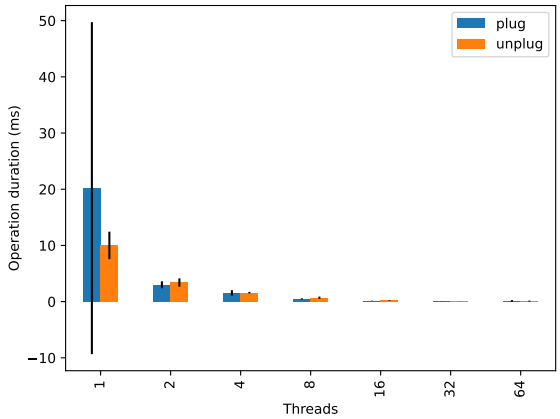


Figure 8.2: Scalability of plug operation

Observation. In Figure 8.2, we can observe that plug and unplug operations can complete in less than 30 ms. We observe very large variation when the number of thread is low which is still unexplained at this stage.

Interpretation. Plug and unplug operation in ExoVM are still dependent of an internal mechanisms which introduces large variation in operation delays. However, we expect the duration of plug and unplug operation is lower than existing hypervisor hotplug techniques for two reasons. First, the bypass of guest page allocator supports saving a few milliseconds. Second, unplugging can be completed quickly as it does not require page migration thanks to the use of 1:1 mappings.

In this section, we have covered the main implementation details of ExoVM. Many part of the implementation are still ongoing work and we still need further

evaluations to validate some of the implementation choices we made to implement ExoVM.

8.3 Evaluation

In section 8.2, we have presented the details of the implementation of ExoVM to offer fast dynamic memory management in VMs. In this section, we perform an evaluation of the different functionalities offered by ExoVM.

First, we evaluate the possibility to let a guest initiate inter-VM shared memory segments using ExoVM and propose a use case for a FaaS platform. Second, we present how ExoVM supports fast elastic adaptation of memory capacity.

8.3.1 inter-VM shared memory segment

In this section, our analysis tries to show how ExoVM shared volatile memory can yield significant speed-up to FaaS runtimes while reducing memory consumption. Our scenario illustrates how a single AI model can be shared across various workers to perform inference.

We try to observe the advantages of ExoVM compared to concurrent backend for a FaaS runtime by first observing how it supports a higher number of ***concurrent activation****. First, we discuss how ExoVM helps to reduce the* ***duration*** *of FaaS activations. Then, we present its impact to reduce the* ***init time*** *of container cold starts in FaaS platforms.*

8.3.1.1 Description of the FAAS experiment

Description In this experiment we setup two VMs with 64 vCPUs and 32 GiB memory. Each VM serves as a worker node for a kubernetes cluster. The kubernetes cluster schedules containers on a uniform view of nodes. We deploy an openwhisk [114] FaaS platform on the kubernetes cluster. Openwhisk deploys *control containers* (API gateway, Scheduler, Invoker, Databases, ...) and *execution containers* (user functions). Openwhisk provides an API which enables a user to define and upload a *function* or *action* with a configuration. The action is created but not invoked directly. A user can then send a request to invoke the action which is named an *activation* in Openwhisk. In our experiment, the action used is an inference phase on a 3.92 GiB LLaMa [144] model.

The FaaS load injector In this experiment, we use FaasLoad [107], a load injector for openwhisk. We configure faasload to use an inter-invocation time of 10 s for 100 activations. This corresponds to the average delay between activations.

We also rely on OpenStack Swift Object Storage [115] used for storage persistency. It is ran in a container in the host. Swift is used on top of a ext4 fileystem using HDD or NVMe as a storage backend. Swift is carefully configured to use 100 workers to prevent swift number of workers to be a bottleneck in the evaluation.

The different backends We review five different backends to run LLaMa inference phase in a function.

First backend is *disk*, it uses a HDD as an underlying storage for Swift storage and K8s cluster. Openwhisk function requires downloading the model locally before executing it with disk backend for every new activation.

Second backend is *NVMe*, it uses a "Intel DC P3700" SSD NVMe with 31 hardware queue pairs (submission queues and completion queues). Openwhisk function requires downloading the model locally before executing it with NVMe backend for every new activation.

Third backend is *ExoVM* which uses a memory segment shared between virtual machines with a preloaded LLaMa model of 3.92 GiB exposed as a DAX device in the guest. It can perform direct memory accesses and requires only a single function to download the model for all subsequent activations.

Fourth backend is *virtiofs* which uses virtiofs [152] a shared file system between virtual machines. Since the filesystem is shared by all virtual machines, the model needs to be downloaded a single times as well.

Fifth backend is *virtiofs-dax* which uses virtiofs [152] a shared file system between virtual machines exposed as a DAX device which is comparable to ExoVM. virtiofs with DAX support is experimental but enables to bypass guest page cache to directly use host page cache. Since the filesystem is shared by all virtual machines, the model needs to be downloaded a single times as well.

The disk backend has been tested but because it is sequential, and FaaS is parallel by definition, it causes activation timeouts on every new run of the benchmark. A takeaway about this evaluation is that a FAAS platform requires a backend supporting parallel IOs.

8.3.1.2 Analysis of activation concurrency

In this experiment, we try to determine how ExoVM can help support a higher number of simultaneous inference phases on a machine. We define concurrency as the number of activations ongoing at each instant in time.

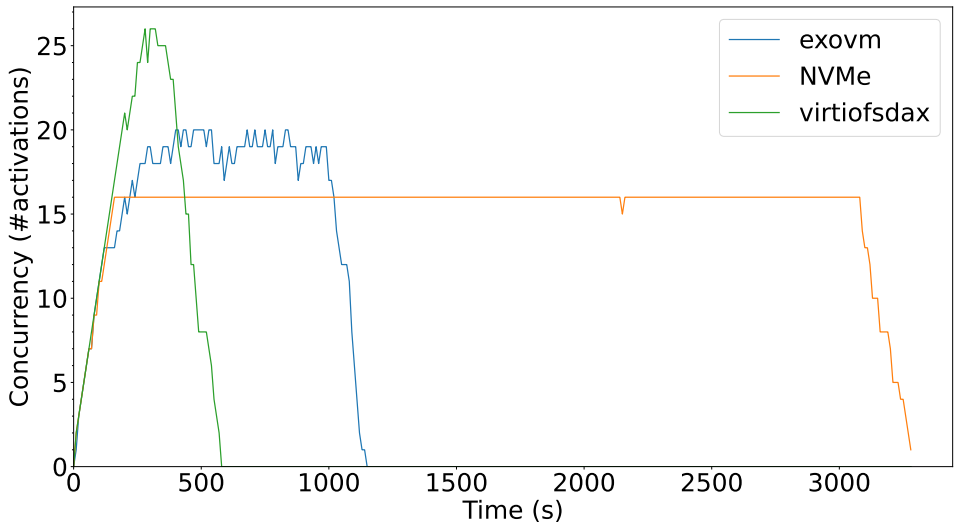


Figure 8.3: Activation concurrency

Description In this experiment, we use FaasLoad to inject new activations in the openwhisk platform. The openwhisk platforms ships a scheduler responsible for scaling the number of concurrent activations based on the available resources. Based on FaasLoad injection policy (inter-invocation time of 10 s for 100 activations), we observe the number of concurrent activations in the platform over time.

Observation Figure 8.3 represents the number of concurrent activations. We can observe that virtiofsdax achieves up to 25 concurrent activations for the smallest duration of execution. ExoVM achieves up to 20 concurrent activations for a total execution duration of 1150 s. NVMe is bottlenecked at 16 concurrent activations for a duration of 3300 s.

Interpretation First, ExoVM reduces total execution time by 3 compared to a NVMe backend. Since, it uses a single 4 GiB shared segment for all instances, it manages to save significant amount of memory which allows scaling to a higher number of activations. On the contrary, in the NVMe case, for each activation execution instance memory is consumed to cache filebacked pages of the IO model. This limits the scaling of instances to only 15 concurrent activations.

Second, virtiofsdax seems to outperforms ExoVM both for duration and maximum number of activations. However, the total duration is smaller because the run is interrupted by a silent error after only 32 activations complete out of 100. We are not investigating the bug any further as it is likely caused by virtiofsdax.

8.3.1.3 Analysis of activation init time

One of the main problems of FaaS platforms is caused by the initialization of containers to execute functions (init time). Some techniques support attaching storage with model pre-populated content. In particular, it is possible to load a ML model on attachable storage to reduce init time. We compare the different techniques available to ExoVM.

Description. In openwhisk, "**init time** is the time spent initializing the function. If this value is present, the action required initialization and represents a cold start. A warm activation will skip initialization, and in this case, the annotation is not generated" [11]

Observation. Figure 8.4 reports the distribution of init time. We can observe that distributions are bimodal with a first mode centred at 0 s and a second mode at different values depending on the labels considered. This bimodal aspect is caused by the nature of function starts which can be either cold or warm. A cold start occurs when the Faas platform has no preinitiated instance of the function ready while warm start are a container reuse for a later invocation. By definition, warm start have an init time of 0 s while cold start require some time to prepare the function. In our case, init phase includes loading the LLaMa model.

Second, we can see that cold start mode is lower for ExoVM with 300 ms while virtiofsdax is centred around 450 ms and NVMe around 2000 ms.

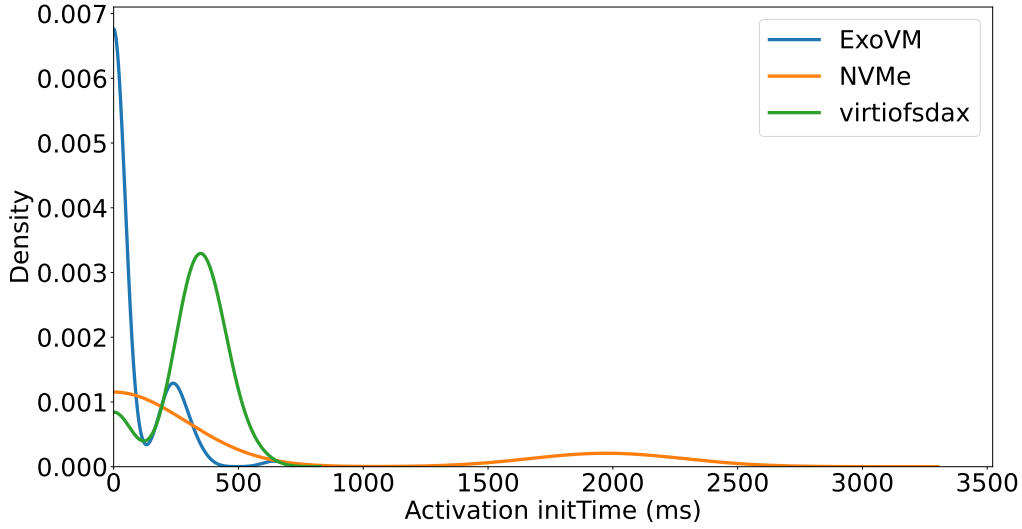


Figure 8.4: Activation init time

Interpretation. ExoVM outperforms the other mechanisms to reduce the init-time of functions.

8.3.1.4 Analysis of activation duration

In the previous experiment, we have focused on the initialization of activations in preparation phases. In this experiment, we try to observe the influence of the backend on the execution time of the function.

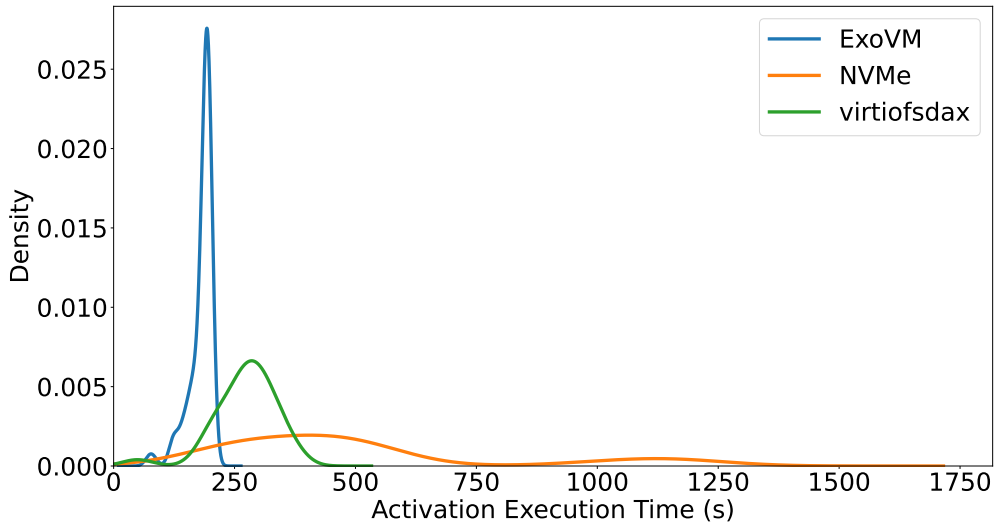


Figure 8.5: FAAS invocations

Observation We can see that ExoVM (190 s) outperforms virtiofsdax (290 s) and NVMe (> 350 s). Additionally, while ExoVM has a narrow distribution centred around its mean value, virtiofsdax and NVMe have broad distribution. NVMe even exhibits a second mode in its distribution.

Interpretation Figure 8.5 shows that outside the init time of container runtime, ExoVM is able to achieve significant performance gain by performing direct access to memory instead of going through file system layers on memory or on a NVMe.

In this section, we have motivated the use of guest-initiated shared memory segments which are supported in ExoVM. Through a detailed evaluation of a FaaS platform, we have exhibited that ExoVM shared memory segments can be used to reduce the initialization of FaaS functions in order to achieve higher concurrency. In particular, we have shown that ExoVM can beat concurrent solutions such as virtiofs-dax or NVMe to share memory between VMs. This evaluation remains a specific use case of ExoVM and the next section will instead focus on how ExoVM is able to support fast memory changes.

8.3.2 Elasticity

In §8.3.1, we have presented the benefits of allocation and plugging of guest-initiated shared memory segments in a Faas runtime. In this section, we present ongoing work on how ExoVM supports fast adjustment of memory capacity in VMs. In particular, we present one of the limit of our prototypes encountered when the VM executes processes which have very short lifetime which stresses the time contribution of plug and unplug operations.

8.3.2.1 Short-lived process and the overhead ExoVM

In this experiment, we try to estimate the performances of ExoVM for short-lived processes. Indeed, ExoVM requires plugging memory when a new process segment is created or extended and unplugging memory when the segment is destroyed (e.g. at process termination). This instrumentation incurs an extra overhead compared to simple process segment creation and destruction used in traditional VMs.

Description In this experiment, we compare tree different VM configurations.

First, we test ExoVM which uses a VM initiated with 2GiB of memory and 64 vCPUs. We use a small instrumentation program which makes compilation transparent by instrumenting mapping of anonymous memory with calls to ExoVM API to hot-add memory in the VM.

Second, we test *Linux 5G*, a VM which is statically allocated with a memory size of 5 GiB and 64 vCPUs.

Third, we test *Linux/FPR 5G*, which is similar to *Linux 5G* but uses the free-page-reporting mechanism (see details in §7.1.2) to report free pages during the VM execution.

We evaluate these configurations with the compilation of Linux Kernel 6.1 with a default kernel configuration. Compilation uses 64 threads with gcc compiler.

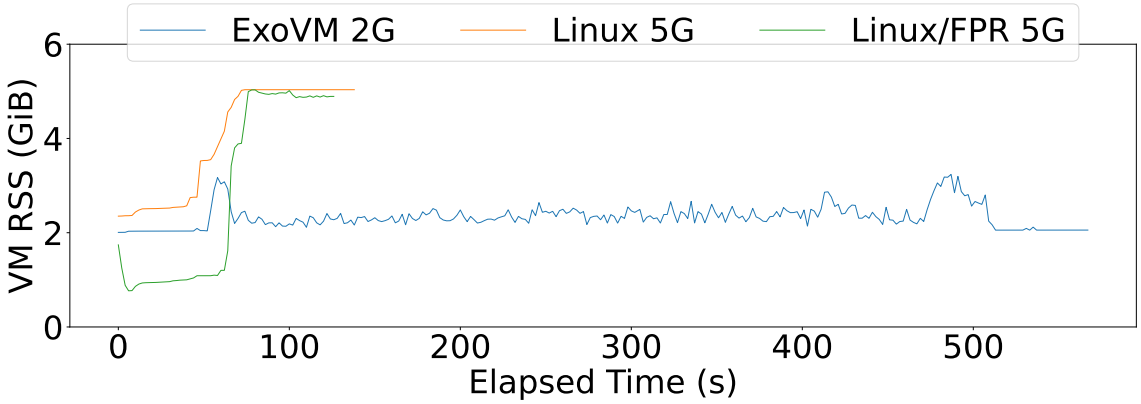


Figure 8.6: Comparison of VM RSS for different techniques

Observation In Figure 8.6, we can observe that the resident set size of the VM oscillates around 2 GiB memory for a duration of 450s.

In static VM (*Linux 5G*), the VM RSS shortly reaches the capacity of the VM (5 GiB) and the memory usage never decreases until the termination of the VM, 100 s later.

In FPR VM (*Linux/FPR 5G*), the VM RSS is only decreased of a few MiB while the usage remains steady around the 5GiB capacity for a duration of 100s. It is not illustrated on this plot, but the RSS in this configuration decreases as soon as the compilation process terminates.

Interpretation Because of the additional blocking time caused by waiting for plug to complete, we were expecting ExoVM to slow down Kernel compile. Indeed, for the compilation of each individual C file, a new process with very short lifetime is created and wait for plug to complete. However, the overhead introduced by ExoVM is much larger than expected and, in total, slows down the compilation by **5**.

During the execution of the program, we have tried to trace the time contributions behind the overhead in ExoVM. We have observed unusually long delays in plug time which prevents the execution of more compilation processes. Normal plug delays cost up to 30 ms, but we have observed up to 2 s spent trying in plug time. We have been able to track down the overhead to long wait time spent trying to acquire a per-CPU read write semaphore mem_hotplug_begin() used in the unplug and plug path in the kernel code. We are currently investigating the reasons behind the long plug delays.

As a summary, in §8.3.1, we have shown the interest of using shard memory segments across VMs support reference to identical pages between VMs. In §8.3.2, we have shown that ExoVM enables to reduce the memory consumption of VMs thanks to quick adaptation of its capacity but ExoVM does not work well with short-lived processes. We leave for future work the evaluation of elasticity in long-lived processes as well as the evaluation of the revocation mechanisms in a collaborative guest which implement elegant revocation of memory region. We also want to evaluate practical scenario showing that abort mechanism is able to maintain safe execution in all VMs

despite an uncollaborative guest.

This chapter presents our second prototype, ExoVM, still under development. ExoVM supports fast adaptation of VM memory capacity to support safe use of dynamic VMs in datacenters.

In details, the main contributions in ExoVM are the following.

The main idea proposed in ExoVM is to let guests report the capacity changes desired instead of detecting it in the hypervisor. Thus, ExoVM proposes to hand to the guest the decisions of issuing memory allocations, freeing, plugging and unplugging. This proposal is based on the idea that collaborative guests should benefit from unused memory in the server. We maintain in the hypervisor the responsibility of verifying when resources should be reclaimed in a guest, as only the hypervisor is aware of memory usage and capacities across all VMs. However, we directly hand to guest applications the responsibility to perform memory reclamation as it allows each guest application to perform finer-grained decisions for memory reclamation. Guest application reclamation is still under progress but we expect faster migration of memory to support fast memory unplug.

Second, since some guests may be uncollaborative and may try to acquire all resources, we propose, as in the exokernel design, an abort protocol to revoke the ability of a guest to acquire hypervisor resources. The implementation and evaluation of this protocol is still under progress.

Third, we implement support for inter-VM memory sharing which is provided simply by decoupling allocation from plug operation.

Fourth, ExoVM draws from the detailed analysis of bottlenecks in existing mechanisms which try to support dynamic capacity changes (memory ballooning, memory hotplug, ...). Notably, ExoVM supports faster plug and unplug operations by bypassing the guest page allocator thanks to the use of 1:1 mappings.

The early evaluation of ExoVM presents the potential use cases enabled by guest-initiated shared memory segments across multiple virtual machines. ExoVM show promising results to reduce memory usage but it still requires improvement to reduce the performance degradation in guest applications. Our ongoing work to support transparent execution of applications with ExoVM should ease the evaluation of many more applications.

Conclusion

Summary

System virtual machines have been in use in datacenters for a couple of decades now. They are at the core of sustainable profits and limited energy consumption in datacenters by supporting resource sharing while guaranteeing isolation between customers. In this thesis, we propose three contributions to reduce memory usage in virtual machine platforms.

First, we propose **ODswap**, a solution to support transparent memory accesses on remote machines using RDMA. ODswap proposes allocation and freeing of remote memory to best use memory leftovers in various servers. ODswap implements on-demand memory consumption on the remote memory server to reduce memory footprint. It also targets the problem of uncollaborative memory management decision by leveraging guest swapping decisions rather than host swapping decisions as in existing prototypes. We perform a detailed evaluation of ODswap where we show that ODswap can outperform DSM-VMs by a factor of 3 to 6 for representative cloud applications.

Second, we present multiple detailed evaluation results to motivate the co-design of VMs and hypervisors. In these evaluations, we present two sets of arguments around **heterogeneity** and **dynamic memory changes**. In a first set of evaluation, we try to determine the reasons behind the widespread use of static VM and the low enthusiasm for techniques supporting dynamic memory capacity changes. We demonstrate that existing techniques support dynamic memory change at the cost of very long delays which almost always lead to guest memory reclamation and may result in undesirable guest process kills. Furthermore, we show that these techniques are implemented in the hypervisor and impose an additional delay caused by detection of desired guest memory changes in feedback control algorithms. In a second set of evaluation, we present the performance impact of information loss between guest memory management and hypervisor memory management. We illustrate the problems of duplicating memory management in two scenarios: **swapping** and **vNUMA topology configuration**. In a first scenario, we show that swapping loses track of metadata information associated with an OS page which results in 100% overhead in IO intensive applications because of uncollaborative decisions. In a second scenario, we show that exposing NUMA topology to the guest with appropriate policy selection can lead to up to 30% performance gains in representative

applications.

Third, we propose **ExoVM**, a work-in-progress prototype which implements fast dynamic memory changes in VMs inspired by exokernel OS design. ExoVM is designed by trying to address the observations presented in our second contribution. ExoVM proposes guest-initiated memory hot-plug and hot-unplug. Additionally, ExoVM decouples plugging operation from resource allocation and unplugging from resource freeing. In addition, this decoupling allows us to support easy memory sharing across VMs and support to let the guest explicitly request memory from specific backends, with desired page granularity. We provide early evaluation of our current prototype to show the potential benefits of shared memory initiated by the guest compared to other prototypes as well as memory savings thanks to dynamic allocation and freeing of resources.

Future work

We consider the following future works on ExoVM which is still unfinished.

Benchmarking revocation of memory in ExoVM. As ExoVM directly let guests manage their memory capacity, we have seen that it requires a revocation mechanism implemented in the hypervisor to revoke the access to memory. We still need to evaluate how fast memory can be revoked and wether it may lead to unsafe scenario. This is our top priority.

Benchmarking elasticity in ExoVM. Currently, we only have limited evaluation results of ExoVM ability to make VM elastic. In particular, our only evaluation is performed on an unrealistic workload based on Linux Kernel compilation. We would like to evaluate ExoVM on more realistic cloud applications. This requires making ExoVM memory management more transparent for applications.

Arbitration in ExoVM. In dynamic memory management solutions, such as ballooning or ExoVM, there exist a stable configuration which happens when the host is under memory pressure where VMs tend to maintain steady capacities in a selfish way. When static VMs are used, hypervisor memory is shared in proportion defined by allocated memory. It can also be seen as, VMs get assigned a weight which corresponds to the ratio of allocated memory over hypervisor memory. In ExoVM, we need to assign a similar weight to VMs to be able to balance memory between VMs when hypervisor memory is missing.

Speeding up ExoVM mechanism. In the stable configuration presented in the above program, guests and hypervisor experience memory pressure. We expect existing techniques which rely on hypervisor page allocator to perform even slower under this configuration. Indeed, in ballooning and memory hotplug, VMs perform unnecessary round-trips to the hypervisor page allocator to release resources which are likely to be allocated again for another VM soon after release time. We would like to confirm our hypothesis with further evaluations and leverage ExoVM design to

speedup borrowing memory. In ExoVM, thanks to the explicit control by the guest of allocation and freeing of memory, it may be possible to implement faster sharing of memory between two guests by delegating page freeing and page allocation to hypervisor user-space.

Side contributions

During this thesis, I have also contributed to various external projects which have inspired new ideas.

JNVM

In particular, in JNVM [91], I performed evaluations to exhibit that go-pmem garbage collector which implements a concurrent mark-sweep does not scale to collect large heaps, in particular on NVDIMM. This observation has motivated the idea to use processing power on remote memory nodes to assist language runtime in the collection of unreferenced objects.

Garbage Collection on disaggregated memory

For a moment, there has been an idea by people working on disaggregated memory that cache coherent interconnects would not scale and would require separating servers in different cache coherency domains. Based on this idea, we have started to work on the possibility of garbage collector to collect memory on uncoherent heap memory. This led to the introduction of the idea of collection on a heap snapshot in remote memory. The control of how and when data should be written back on remote memory is supported by the introduction of a software cache which enables to instrument writeback. Instead of synchronizing mutator threads and collector threads on every write, using a write barrier mechanism, we propose to leverage the software cache and to perform synchronization only during writeback. Since writeback operations occur less frequently than write operations, this design could reduce the cost of barrier for synchronization in concurrent GC. Implementation and correction to the initial design work have been made during a 6 month internship by Adam Chader which carried on the work in a PHD thesis.

Abusing Dune for memory placement

As presented in §4.3.4, various works [60, 125] have shown that swap-based prototypes introduces multiple biases causing poor simulation conditions to remote memory accesses. The problem is that alternatives to swapping for remote memory

access such as direxttCXL [60] require hardware support. Other alternatives like AIFM [125] require modifications to source code to perform remote memory accesses. Thus, we have tried to look for alternatives to swapping to support transparent placement of pages on memory tiers with per-application policy. We found in Dune [20] a relevant method to implement transparent tiering mechanisms directly in the application. Indeed, by abusing virtualization hardware extensions, Dune let processes directly manage their own page table. In particular, Dune maintains physical memory integrity and confidentiality by leveraging the second level page table (EPT) to really isolate processes. Dune still suffers from IO amplification as it manages pages instead of objects and it still relies on page fault management which prevents CPU pipelining. However, custom placement policies implemented in each application could lead to sufficient improvement and prevent the drawbacks of Dune. This idea has led to a master project, followed by a PHD thesis, conducted by Jana Tolga to implement a runtime where transparent tiering policies can be implemented.

Appendix

All graphs presented in this chapter present call stacks from top to bottom in a graph named icycle graph. The width of frames represents the time contribution of the different functions. Saturated colors also represent the largest time contributions.

Memory ballooning

Inflate command

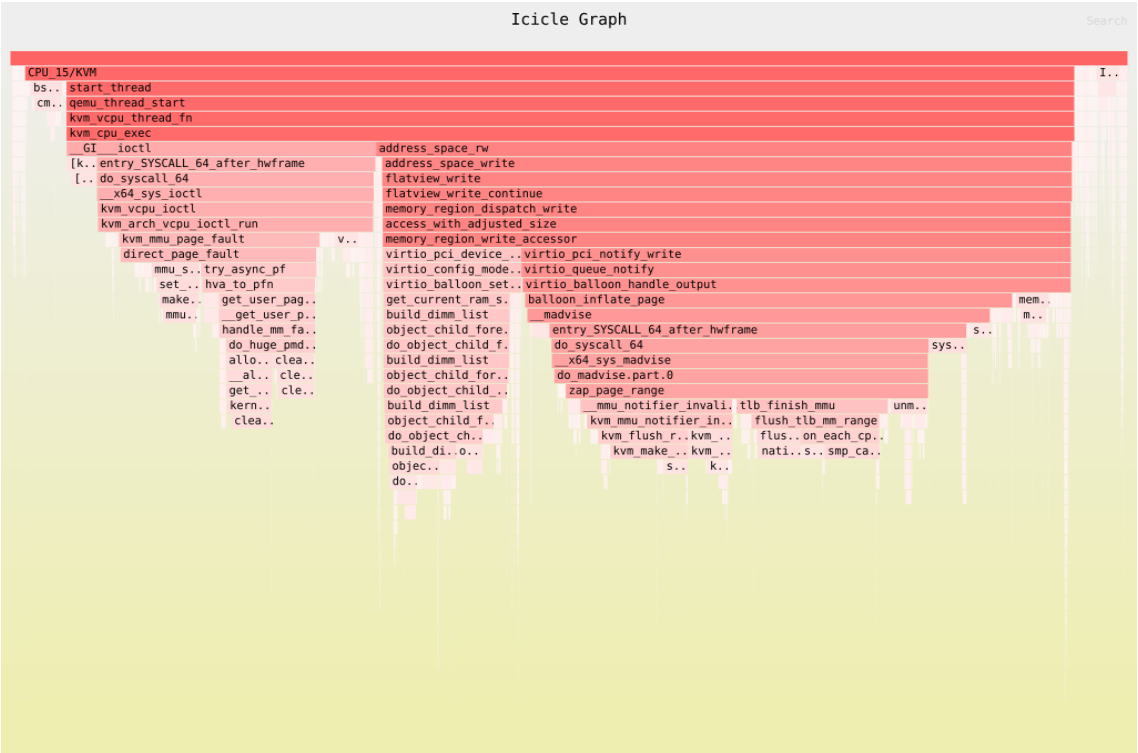


Figure 7: Hypervisor inflate ICycle Graph

Figure 7 shows three independent contributions to CPU time during the measurement.

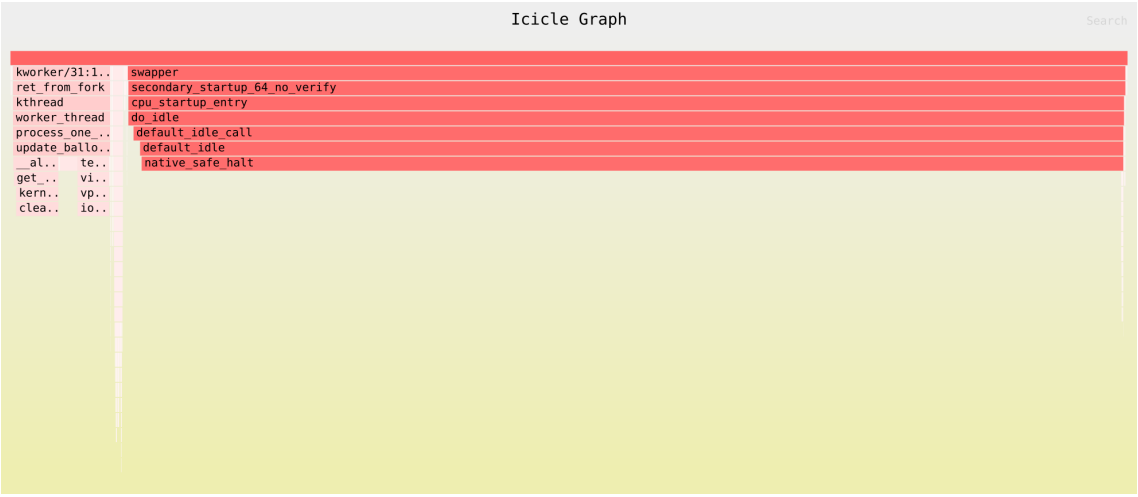


Figure 8: Guest inflate ICycle Graph

First, the top bar in the histogram reports that a single thread among the 16 threads used for the VMs has a large CPU usage.
Second, the leftmost frame under qemu kvm_cpu_exec() stands for 27.7 % of total time. These frames represent calls to qemu kvm_mmu_page_fault() which is the time spent handling two-dimensional page fault (hypervisor page fault) and the resulting host OS page fault associated.
The graph is then split in two halves under qemu memory_region_write_accessor() which represents 61.60 %. The first half is the callstack for qemu virtio_pci_device_write() which represents 13% of the total time. This method is used to write to PCIe MMIO configuration space of guest device from the host to trigger the inflation command.
The other half is the callstack to qemu virtio_pci_notify_write() which accounts for 41.42 % of the total inflation time. This method handles reception of a notification in the host of a guest write to the MMIO region. The method implements the host side part of the inflation inside qemu. All the time-contribution is spent on calling madvise() system call with DONT_NEED argument to perform **freeing of pages**. Kernel side of madvise (32.43 %) spends its time in zap_page_range() with 13 % spent in **kernel MMU notification system** and ensure coherence of EPT and hypervisor page table and 13% in TLB shootdown.

Figure 8 shows three main contributions for the single-threaded inflation worker in the guest under update_balloon_size_func. First, __alloc_pages_nodemask() accounts for 46 % of the inflation time spent retrieving pages from buddy free lists with expensive zeroing of pages (clear_page_rep()).
Second, 18 % of inflation time is spent in MMIO write to report the number of pages (iowrite32). This traps to qemu MMIO callbacks which explains the high cost.
Third, tell_host() which performs the notification from the guest to the host represents 33 % of the inflation time.

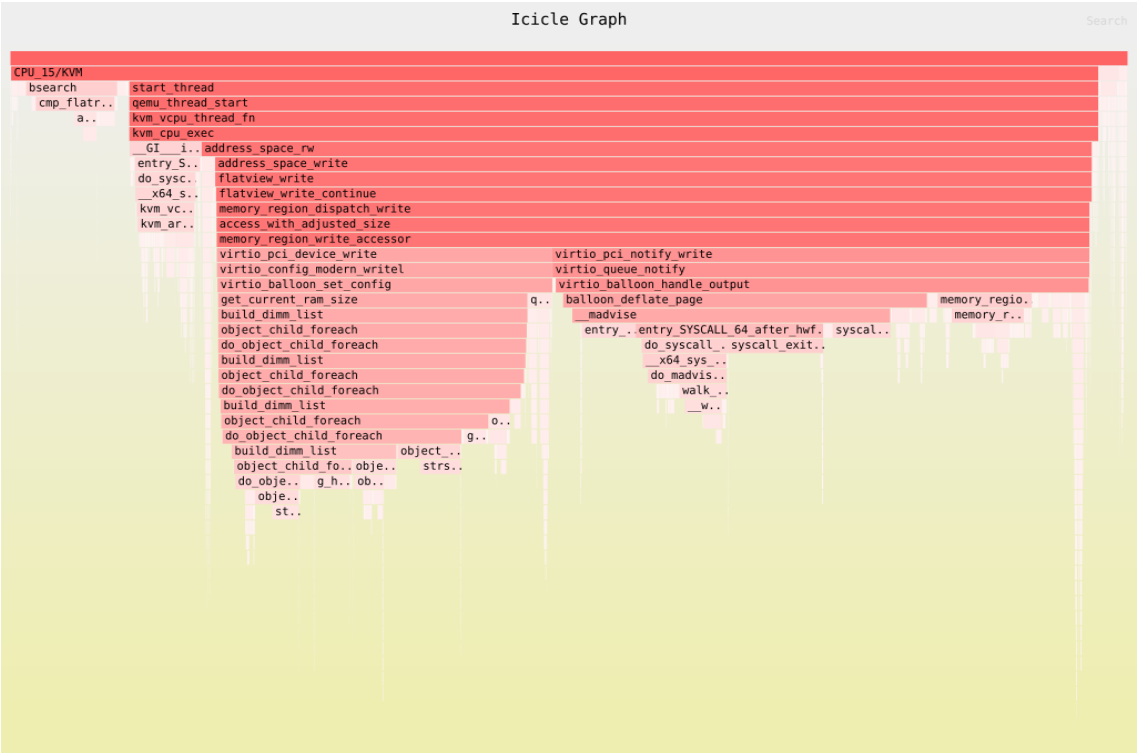


Figure 9: Hypervisor deflate ICycle Graph

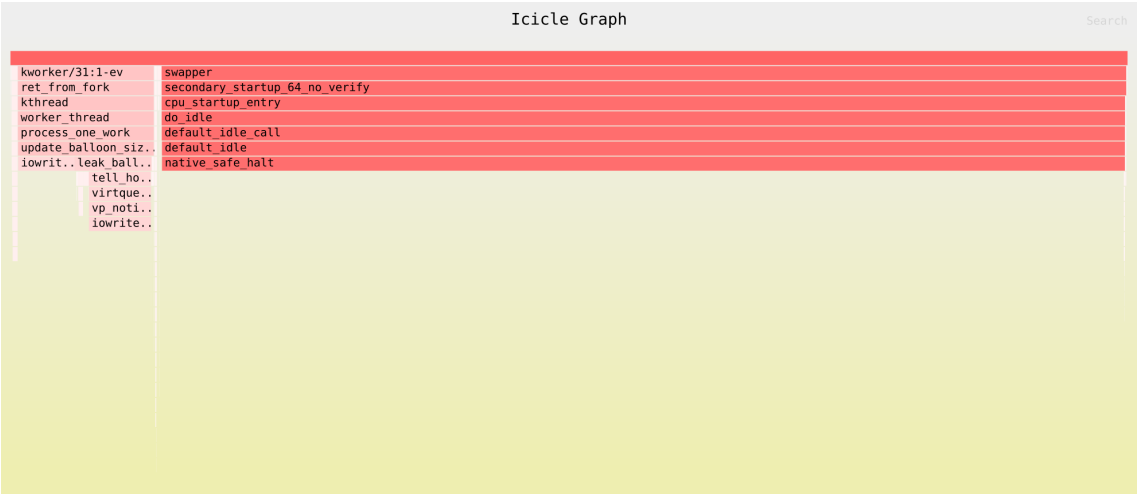


Figure 10: Guest deflate ICycle Graph

Deflate command

Figure 9 shows two independent contributions to CPU time during the measurement. First, as in hypervisor side of balloon inflation, we can see that a single thread is used over the 16 threads allocated for VM vCPUS. Second, the leftmost part under qemu kvm_cpu_exec() for MMIO emulation accounts for 6 % of total time. virtio_pci_device_write() which corresponds to MMIO writes to configuration space accounts for 30% of total CPU time. Third, the rightmost part for virtio_pci_notify_write which implements the deflation inside qemu with a call to

madvise(WILL_NEED) accounts for 32% of total CPU time.

Figure 10 shows two independent contributions. First, 41 % of time is spent in MMIO write (iowrite32). Second, 56 % of the deflate time is spent dequeuing pages from balloon list and freeing pages and notifying the host.

Paravirtual memory hotplug

Unplug

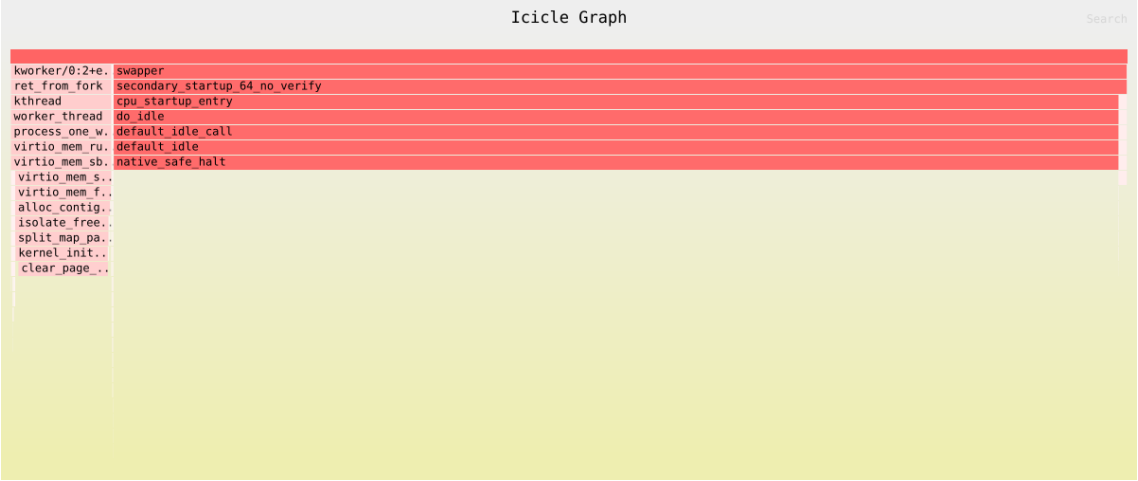


Figure 11: Guest virtio-mem unplug ICycle Graph

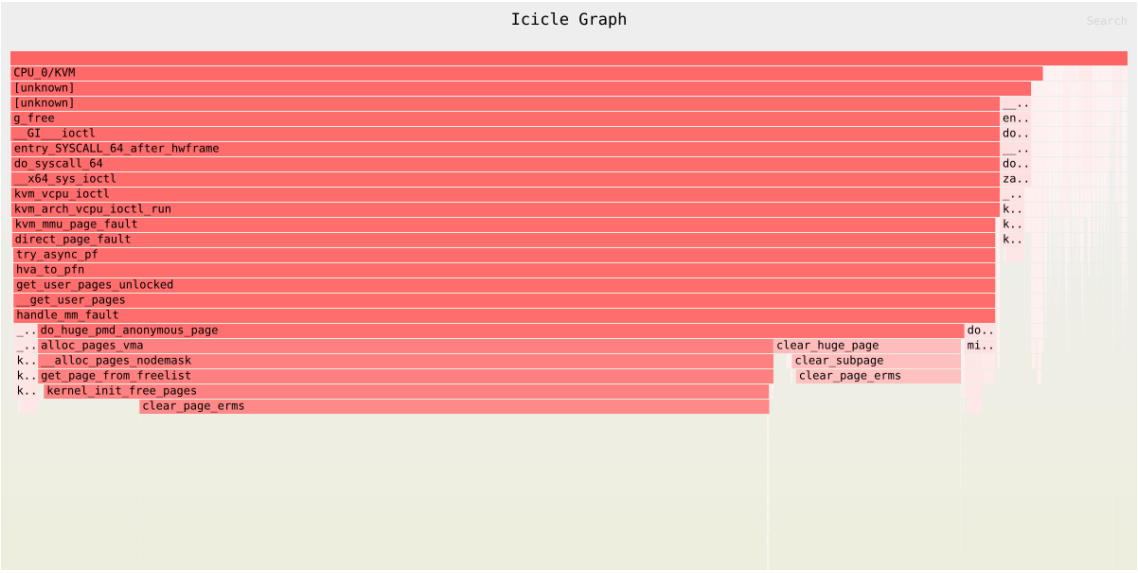


Figure 12: Hypervisor virtio-mem unplug ICycle Graph

Observation Figure 12 and Figure 11 report CPU time contribution of 64 GiB in hypervisor and guest OS respectively. We observe that guest OS performs

monothreaded defragmentation in virtio _mem_run_wq workqueue with 4.2% of time trying to physically offline and remove memory (offline _and_remove_memory) and 95.7% in fake offlining (virtio _mem_fake_offline). Real offlinning and removal of memory is split in offlining memory as a device (try_ offline _memory_block()) which walks through memory blocks to offline them which accounts for 3.8% and removal of pages from the buddy allocator (try_remove_memory()) which accounts for the remaining 0.3%. In order to logically offline pages, virtio-mem needs to allocate pages on the desired range. It isolates free pages in a freelist and split them in order 0 pages before setting PG_Offline page flag. This allocation scheme leads to page zeroing which accounts for 89% of the total unplug time. In Figure 12, we observe that page zeroing at guest level leads to expensive hypervisor EPT page fault handling with page allocation also requiring to zero pages at different levels of the callstack with 80% of page fault handling time spent in page zeroing.

Plug

Observation Figure 14 and Figure 13 report CPU time contribution for hot-plug operation of 64 GiB in hypervisor and guest OS respectively. It is important to keep in mind that plug operation is very short compared to hotunplug. Moreover, the scenario we are studying present the case where virtio-mem will need to physically unplug vDIMMs and does not just perform logical offlining of vDIMMs. First, we can see that hot-adding of memory in guest OS is performed in a workqueue which spend 98% of the guest OS time onlining pages (online_pages()), 1.5% of time adding pages to the buddy allocator (add_pages(), initiating memory devices and physical memory mapping. During online, 33.5% of the time is spent in initializing memmap by setting page refcount to 1 and marking pages as reserved (memmap_ init_zone()). Then, onlining checks if a zone contains holes (set_zone_contiguous()) which accounts for 17 % of the time. At this stage, pages are still isolated, thus onlining calls undo_isolate_page_range() to activate allocation on these pages. This method may spin trying to acquire the zone spinlock. In our case, we have observed 12% CPU time consumed in this method. Finally, 34.4 % is clearing reserved bit in page metadata and dropping refcount to 0 and adding pages to buddy free-lists by calls to free_one_page(). Hypervisor time contributions are negligible to be reported.

Interpretation Most of the time is spent by guest OS zeroing pages when it tries to isolate pages to logically set them as offline. Interestingly, guest zeroing leads to hypervisor EPT fault handling with page allocation and additional page zeroing. Hypervisor page zeroing is required to prevent information leakage to the guest.

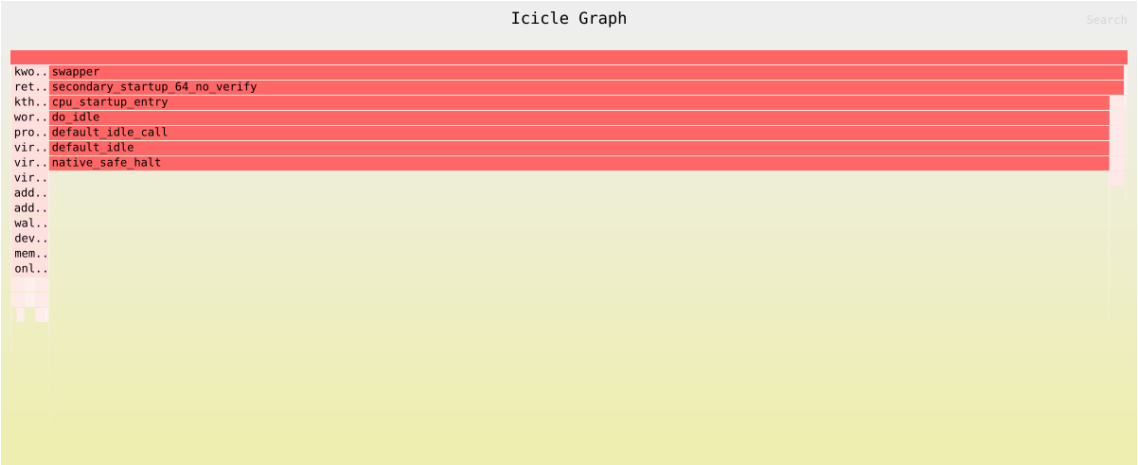


Figure 13: Guest virtio-mem plug ICycle Graph

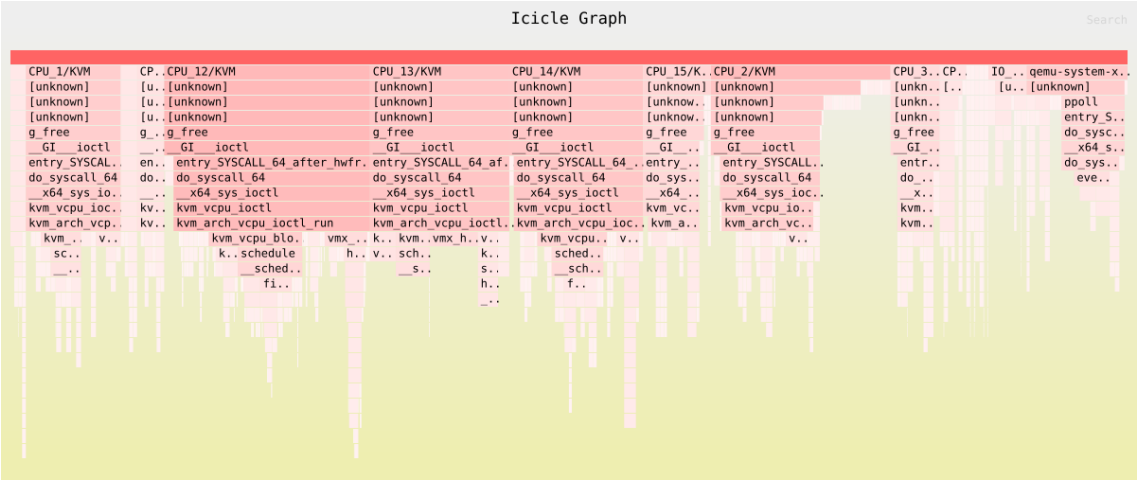


Figure 14: Hypervisor virtio-mem plug ICycle Graph

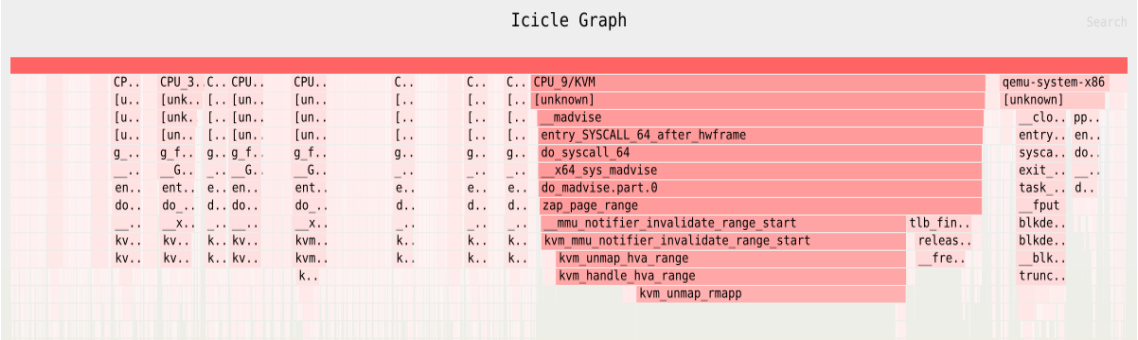


Figure 15: Hypervisor report ICycle Graph

Bibliography

[1] D. Abramson et al. *Intel Virtualization Technology for Directed I/O*. 2006.

[2] *ACPI Specification - System Address Map Interfaces*. Jan. 2021. URL: https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/15_System_Address_Map_Interfaces/Sys_Address_Map_Interfaces.html.

[3] Alexandru Agache et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[4] Anup Agarwal et al. "Unlocking unallocated cloud capacity for long, uninterruptible workloads". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 457–478. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/agarwal-anup.

[5] Marcos K. Aguilera et al. "Designing Far Memory Data Structures: Think Outside the Box". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: Association for Computing Machinery, 2019, pp. 120–126. ISBN: 9781450367271. DOI: 10.1145/3317550.3321433. URL: https://doi.org/10.1145/3317550.3321433.

[6] Shoaib Akram et al. "Write-Rationing Garbage Collection for Hybrid Memories". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 62–77. ISBN: 9781450356985. DOI: 10.1145/3192366.3192392. URL: https://doi.org/10.1145/3192366.3192392.

[7] Emmanuel Amaro et al. "Can Far Memory Improve Job Throughput?" In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387522. URL: https://doi.org/10.1145/3342195.3387522.

[8] Nadav Amit, Dan Tsafrir, and Assaf Schuster. "VSwapper: A Memory Swapper for Virtualized Environments". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 349–366. ISBN: 9781450323055. DOI: 10.1145/2541940.2541969. URL: https://doi.org/10.1145/2541940.2541969.

[9] Nadav Amit et al. "VIOMMU: Efficient IOMMU Emulation". In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'11. Portland, OR: USENIX Association, 2011, p. 6.

[10] *An Introduction to the Intel ©QuickPath Interconnect*. 2009. URL: https://www.intel.ca/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

[11] *Annotations on OpenWhisk assets*. URL: https://github.com/apache/openwhisk/blob/master/docs/annotations.md.

[12] *Apache Hadoop*. URL: https://hadoop.apache.org/.

[13] *Apache Mesos*. URL: https://mesos.apache.org/.

[14] Andrea Arcangeli. *AutoNUMA*. May 2012. URL: https://mirrors.edge.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf.

[15] *autonuma: reduce cache footprint when scanning page tables*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a818f5363a0eba04bcff986c64c919d3f44b8017.

[16] Vlastimil Babka. *Overview of Memory Reclaim in the Current Upstream Kernel*. 2021. URL: https://lpc.events/event/11/contributions/896/attachments/793/1493/slides-r2.pdf.

[17] Vlastimil Babka. *The hard work behind large physical memory allocations in the kernel*. 2023. URL: https://lpc.events/event/2/contributions/65/attachments/15/171/slides-expanded.pdf.

[18] D.H. Bailey et al. "The Nas Parallel Benchmarks". In: *Int. J. High Perform. Comput. Appl.* 5.3 (Sept. 1991), pp. 63–73. ISSN: 1094-3420. DOI: 10.1177/109434209100500306. URL: https://doi.org/10.1177/109434209100500306.

[19] Paul Barham et al. "Xen and the Art of Virtualization". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 164–177. ISBN: 1581137575. DOI: 10.1145/945445.945462. URL: https://doi.org/10.1145/945445.945462.

[20] Adam Belay et al. "Dune: Safe User-level Access to Privileged CPU Features". In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 335–348. ISBN: 978-1-931971-96-6. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay.

[21] Anton Beloglazov and Rajkumar Buyya. "Managing Overloaded Hosts for Dynamic Consolidation of Virtual Machines in Cloud Data Centers under Quality of Service Constraints". In: *IEEE Transactions on Parallel and Distributed Systems* 24.7 (2013), pp. 1366–1379. DOI: 10.1109/TPDS.2012.240.

[22] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, Jan. 2011.

[23] Matias Bjørling et al. "Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems". In: *Proceedings of the 6th International Systems and Storage Conference*. SYSTOR '13. Haifa, Israel: Association for Computing Machinery, 2013. ISBN: 9781450321167. DOI: 10.1145/2485732.2485740. URL: https://doi.org/10.1145/2485732.2485740.

[24] D. L. Black et al. "Translation Lookaside Buffer Consistency: A Software Approach". In: *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS III. Boston, Massachusetts, USA: Association for Computing Machinery, 1989, pp. 113–122. ISBN: 0897913000. DOI: 10.1145/70082.68193. URL: https://doi.org/10.1145/70082.68193.

[25] Jeff Bonwick. "The Slab Allocator: An Object-Caching Kernel". In: *USENIX Summer 1994 Technical Conference (USENIX Summer 1994 Technical Conference)*. Boston, MA: USENIX Association, June 1994. URL: https://www.usenix.org/conference/usenix-summer-1994-technical-conference/slab-allocator-object-caching-kernel.

[26] Edouard Bugnion et al. "Disco: Running Commodity Operating Systems on Scalable Multiprocessors". In: *ACM Trans. Comput. Syst.* 15.4 (Nov. 1997), pp. 412–447. ISSN: 0734-2071. DOI: 10.1145/265924.265930. URL: https://doi.org/10.1145/265924.265930.

[27] Qingchao Cai et al. "Efficient Distributed Memory Management with RDMA and Caching". In: *Proc. VLDB Endow.* 11.11 (July 2018), pp. 1604–1617. ISSN: 2150-8097. DOI: 10.14778/3236187.3236209. URL: https://doi.org/10.14778/3236187.3236209.

[28] Irina Calciu et al. "Rethinking Software Runtimes for Disaggregated Memory". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 79–92. ISBN: 9781450383172. DOI: 10.1145/3445814.3446713. URL: https://doi.org/10.1145/3445814.3446713.

[29] Blake Caldwell et al. "FluidMem: Full, Flexible, and Fast Memory Disaggregation for the Cloud". In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 2020, pp. 665–677. DOI: 10.1109/ICDCS47774.2020.00090.

[30] Ho-Ren Chuang et al. “Aggregate VM: Why Reduce or Evict VM’s Resources When You Can Borrow Them From Other Nodes?” In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys ’23. Rome, Italy: Association for Computing Machinery, 2023, pp. 469–487. ISBN: 9781450394871. DOI: 10.1145/3552326.3587452. URL: https://doi.org/10.1145/3552326.3587452.

[31] Christopher Clark et al. “Live Migration of Virtual Machines”. In: *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation - Volume 2*. NSDI’05. USA: USENIX Association, 2005, pp. 273–286.

[32] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. “Scalable Address Spaces Using RCU Balanced Trees”. In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: Association for Computing Machinery, 2012, pp. 199–210. ISBN: 9781450307598. DOI: 10.1145/2150976.2150998. URL: https://doi.org/10.1145/2150976.2150998.

[33] *Coherent Device Attribute Table (CDAT) Specification*. Oct. 2020. URL: https://uefi.org/sites/default/files/resources/Coherent%20Device%20Attribute%20Table_1.01.pdf.

[34] *Compute Express Link (CXL) Specification*. Aug. 2022. URL: https://www.computeexpresslink.org/.

[35] F.J. Corbató and Project MAC (Massachusetts Institute of Technology). *A PAGING EXPERIMENT WITH THE MULTICS SYSTEM*. Project MAC. Massachusetts Institute of Technology, 1968. URL: https://books.google.fr/books?id=5wDQNwAACAAJ.

[36] Jonathan Corbet. *Multi-generational LRU: the next generation*. 2021. URL: https://lwn.net/Articles/856931/.

[37] Jonathan Corbet. *Two memory-tiering patch sets*. 2022. URL: https://lwn.net/Articles/898766/.

[38] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. 3rd ed. O’Reilly Media, Inc., 2005. ISBN: 9780596005900.

[39] Eli Cortez et al. “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: Association for Computing Machinery, 2017, pp. 153–167. ISBN: 9781450350853. DOI: 10.1145/3132747.3132772. URL: https://doi.org/10.1145/3132747.3132772.

[40] Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom. “Code Sharing among Virtual Machines”. In: *ECOOP 2002 — Object-Oriented Programming*. Ed. by Boris Magnusson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 155–177. ISBN: 978-3-540-47993-2.

[41] Michael D. Dahlin et al. "Cooperative Caching: Using Remote Client Memory to Improve File System Performance". In: *First Symposium on Operating Systems Design and Implementation (OSDI 94)*. Monterey, CA: USENIX Association, Nov. 1994. URL: https://www.usenix.org/conference/osdi-94/cooperative-caching-using-remote-client-memory-improve-file-system-performance.

[42] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN: 9780080497808.

[43] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 127–144. ISBN: 9781450323055. DOI: 10.1145/2541940.2541941. URL: https://doi.org/10.1145/2541940.2541941.

[44] Umesh Deshpande et al. "MemX: Virtualization of Cluster-Wide Memory". In: *Proceedings of the 2010 39th International Conference on Parallel Processing*. ICPP '10. USA: IEEE Computer Society, 2010, pp. 663–672. ISBN: 9780769541563. DOI: 10.1109/ICPP.2010.74. URL: https://doi.org/10.1109/ICPP.2010.74.

[45] *Device Specification for Inter-VM shared memory device*. URL: https://github.com/qemu/qemu/blob/master/docs/specs/ivshmem-spec.txt.

[46] Aleksandar Dragojević et al. "FaRM: Fast Remote Memory". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 401–414. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%7B%5C'c%7D.

[47] Alexander Duyck. *mm: introduce Reported pages*. 2020. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=36e66c554b5c6a9d17a229faca7a61693527b0bd.

[48] Charles Elkan. "Using the Triangle Inequality to Accelerate K-Means". In: *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*. ICML'03. Washington, DC, USA: AAAI Press, 2003, pp. 147–153. ISBN: 1577351894.

[49] D. R. Engler, M. F. Kaashoek, and J. O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: Association for Computing Machinery, 1995, pp. 251–266. ISBN: 0897917154. DOI: 10.1145/224056.224076. URL: https://doi.org/10.1145/224056.224076.

[50] Dan Ernst. *Follow the Data: Memory-Centric Designs for Modern Datacenters*. Feb. 2023. URL: https://www.youtube.com/watch?v=UA0pLW3QG5c.

[51] Jason Evans. “A Scalable Concurrent malloc(3) Implementation for FreeBSD”. In: (Jan. 2006).

[52] Joshua Fried et al. “Caladan: Mitigating Interference at Microsecond Timescales”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 281–297. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/fried.

[53] Yaosheng Fu, Tri M. Nguyen, and David Wentzlaff. “Coherence Domain Restriction on Large Scale Systems”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: Association for Computing Machinery, 2015, pp. 686–698. ISBN: 9781450340342. DOI: 10.1145/2830772.2830832. URL: https://doi.org/10.1145/2830772.2830832.

[54] Alexander Fuerst et al. “Memory-Harvesting VMs in Cloud Platforms”. In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 583–594. ISBN: 9781450392051. DOI: 10.1145/3503222.3507725. URL: https://doi.org/10.1145/3503222.3507725.

[55] Paul A. Games and John F. Howell. “Pairwise Multiple Comparison Procedures with Unequal N’s and/or Variances: A Monte Carlo Study”. In: *Journal of Educational Statistics* 1.2 (1976), pp. 113–125. DOI: 10.3102/10769986001002113. eprint: https://doi.org/10.3102/10769986001002113. URL: https://doi.org/10.3102/10769986001002113.

[56] Gregory R. Ganger et al. “Fast and Flexible Application-Level Networking on Exokernel Systems”. In: *ACM Trans. Comput. Syst.* 20.1 (Feb. 2002), pp. 49–83. ISSN: 0734-2071. DOI: 10.1145/505452.505455. URL: https://doi.org/10.1145/505452.505455.

[57] Peter X. Gao et al. “Network Requirements for Resource Disaggregation”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 249–264. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao.

[58] Mel Gorman. *Foundation for automatic NUMA balancing*. 2021. URL: https://lwn.net/Articles/523065/.

[59] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. USA: Prentice Hall PTR, 2004. ISBN: 0131453483.

[60] Donghyun Gouk et al. “Direct Access, High-Performance Memory Disaggregation with DirectCXL”. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 287–294. ISBN: 978-1-939133-29-65. URL: https://www.usenix.org/conference/atc22/presentation/gouk.

[61] Juncheng Gu et al. "Efficient Memory Disaggregation with Infiniswap". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 649–667. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[62] Ori Hadary et al. "Protean: VM Allocation Service at Scale". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 845–861. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/hadary.

[63] Thomas Haynes. *Network File System (NFS) Version 4 Minor Version 2 Protocol*. RFC 7862. Nov. 2016. DOI: 10.17487/RFC7862. URL: https://www.rfc-editor.org/info/rfc7862.

[64] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[65] Benedict Herzog et al. "The Price of Meltdown and Spectre: Energy Overhead of Mitigations at Operating System Level". In: *Proceedings of the 14th European Workshop on Systems Security*. EuroSec '21. Online, United Kingdom: Association for Computing Machinery, 2021, pp. 8–14. ISBN: 9781450383370. DOI: 10.1145/3447852.3458721. URL: https://doi.org/10.1145/3447852.3458721.

[66] David Hildenbrand. *virtio-mem: Paravirtualized Memory*. 2018. URL: https://events19.linuxfoundation.org/wp-content/uploads/2017/12/virtio-mem-Paravirtualized-Memory-David-Hildenbrand-Red-Hat-1.pdf.

[67] David Hildenbrand and Martin Schulz. "Virtio-Mem: Paravirtualized Memory Hot(Un)Plug". In: *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 1–14. ISBN: 9781450383943. DOI: 10.1145/3453933.3454010. URL: https://doi.org/10.1145/3453933.3454010.

[68] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. "Post-Copy Live Migration of Virtual Machines". In: *SIGOPS Oper. Syst. Rev.* 43.3 (July 2009), pp. 14–26. ISSN: 0163-5980. DOI: 10.1145/1618525.1618528. URL: https://doi.org/10.1145/1618525.1618528.

[69] Takahiro Hirofuchi and Ryousei Takano. "RAMinate: Hypervisor-Based Virtualization for Hybrid Main Memory Systems". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC '16. Santa Clara, CA, USA: Association for Computing Machinery, 2016, pp. 112–125. ISBN: 9781450345255. DOI: 10.1145/2987550.2987570. URL: https://doi.org/10.1145/2987550.2987570.

[70] Naoya Horiguchi. *hugetlb: hugepage migration core*. 2010. URL: https://git.kernel.org/pub/scm/linux/kernel/git/akpm/mm.git/commit/?h=mm-unstable&id=290408d4a25002f099efeee7b6a5778d431154d6.

[71] *Intel Rack Scale Design (Intel RSD)*. Nov. 2016. URL: https://www.intel.fr/content/www/fr/fr/architecture-and-technology/rack-scale-design-overview.html.

[72] *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3C: System Programming Guide, Part 3*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf.

[73] *iSCSI Extensions for RDMA Specification (Version 1.0)*. July 2003. URL: http://www.rdmaconsortium.org/home/draft-ko-iwarp-iser-v1.PDF.

[74] Joseph Izraelevitz et al. *Basic Performance Measurements of the Intel Optane DC Persistent Memory Module*. 2019. DOI: 10.48550/ARXIV.1903.05714. URL: https://arxiv.org/abs/1903.05714.

[75] Joseph Izraelevitz et al. *Basic Performance Measurements of the Intel Optane DC Persistent Memory Module*. 2019. DOI: 10.48550/ARXIV.1903.05714. URL: https://arxiv.org/abs/1903.05714.

[76] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st. Chapman & Hall/CRC, 2011. ISBN: 1420082795.

[77] Sangeetha Abdu Jyothi et al. "Morpheus: Towards Automated SLOs for Enterprise Clusters". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 117–134. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/jyothi.

[78] Aneesh Kumar K.V. *mm/demotion: add support for explicit memory tiers*. 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/akpm/mm.git/commit/?h=mm-unstable&id=7b3ef2e6a64440924ecbcc5b6d3f8b7966558c66.

[79] M. Frans Kaashoek et al. "Application Performance and Flexibility on Exokernel Systems". In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. SOSP '97. Saint Malo, France: Association for Computing Machinery, 1997, pp. 52–65. ISBN: 0897919165. DOI: 10.1145/268998.266644. URL: https://doi.org/10.1145/268998.266644.

[80] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems". In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. Denver, CO, USA: USENIX Association, 2016, pp. 437–450. ISBN: 9781931971300.

[81] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA Efficiently for Key-Value Services". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 295–306. ISBN: 9781450328364. DOI: 10.1145/2619239.2626299. URL: https://doi.org/10.1145/2619239.2626299.

[82] David Kanter. *The Common System Interface: Intel's Future Interconnect*. Aug. 2007. URL: https://www.realworldtech.com/common-system-interface.

[83] Kim Keeton. *The Machine*. HP. May 2016. URL: https://www.youtube.com/watch?v=0EQY0q4EykY.

[84] Alexey Khrabrov et al. "JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 869–884. ISBN: 978-1-939133-29-62. URL: https://www.usenix.org/conference/atc22/presentation/khrabrov.

[85] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. "Exploring the Design Space of Page Management for Multi-Tiered Memory Systems". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 715–728.

[86] Cristian Klein et al. "Brownout: Building More Robust Cloud Applications". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 700–711. ISBN: 9781450327565. DOI: 10.1145/2568225.2568227. URL: https://doi.org/10.1145/2568225.2568227.

[87] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896834.

[88] Matthew J. Koop et al. "Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate InfiniBand". In: *2008 16th IEEE Symposium on High Performance Interconnects*. 2008, pp. 85–92. DOI: 10.1109/HOTI.2008.26.

[89] Mohan Kumar Kumar et al. "LATR: Lazy Translation Coherence". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 651–664. ISBN: 9781450349116. DOI: 10.1145/3173162.3173198. URL: https://doi.org/10.1145/3173162.3173198.

[90] Seung-seob Lee et al. "MIND: In-Network Memory Management for Disaggregated Data Centers". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 488–504. ISBN: 9781450387095. DOI: 10.1145/3477132.3483561. URL: https://doi.org/10.1145/3477132.3483561.

[91] Anatole Lefort et al. "J-NVM: Off-Heap Persistent Objects in Java". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 408–423. ISBN: 9781450387095. DOI: 10.1145/3477132.3483579. URL: https://doi.org/10.1145/3477132.3483579.

[92] Ilya Lesokhin et al. "Page Fault Support for Network Controllers". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 449–466. ISBN: 9781450344654. DOI: 10.1145/3037697.3037710. URL: https://doi.org/10.1145/3037697.3037710.

[93] Michel Lespinasse. *Speculative page faults*. Apr. 2021. URL: https://lwn.net/Articles/851853/.

[94] G. Ley and D. Phipps. "Design and analysis of a synchronous dram memory module". In: *IEEE International Workshop on Memory Technology, Design and Testing,* 1996, pp. 72–78. DOI: 10.1109/MTDT.1996.782495.

[95] Huaicheng Li et al. *Pond: CXL-Based Memory Pooling Systems for Cloud Platforms*. 2022. DOI: 10.48550/ARXIV.2203.00241. URL: https://arxiv.org/abs/2203.00241.

[96] Kai Li. "IVY: A Shared Virtual Memory System for Parallel Computing". In: *Proceedings of the International Conference on Parallel Processing, ICPP '88, The Pennsylvania State University, University Park, PA, USA, August 1988. Volume 2: Software*. Pennsylvania State University Press, 1988, pp. 94–101.

[97] Heiner Litz et al. "RAIL: Predictable, Low Tail Latency for NVMe Flash". In: *ACM Trans. Storage* 18.1 (Jan. 2022). ISSN: 1553-3077. DOI: 10.1145/3465406. URL: https://doi.org/10.1145/3465406.

[98] David Lo et al. "Heracles: Improving Resource Efficiency at Scale". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. Portland, Oregon: Association for Computing Machinery, 2015, pp. 450–462. ISBN: 9781450334020. DOI: 10.1145/2749469.2749475. URL: https://doi.org/10.1145/2749469.2749475.

[99] Gabriel H. Loh. "3D-Stacked Memory Architectures for Multi-Core Processors". In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA '08. USA: IEEE Computer Society, 2008, pp. 453–464. ISBN: 9780769531748. DOI: 10.1109/ISCA.2008.15. URL: https://doi.org/10.1109/ISCA.2008.15.

[100] Maxime Lorrillere et al. "Puma: Pooling Unused Memory in Virtual Machines for I/O Intensive Applications". In: *Proceedings of the 8th ACM International Systems and Storage Conference*. SYSTOR '15. Haifa, Israel: Association for Computing Machinery, 2015. ISBN: 9781450336079. DOI: 10.1145/2757667.2757669. URL: https://doi.org/10.1145/2757667.2757669.

[101] *Lustre distributed filesystem*. URL: git://git.whamcloud.com/fs/lustre-release.git (visited on 02/28/2023).

[102] Shirley Ma. *vhost: vhost TX zero-copy support*. 2011. URL: https://git.kernel.org/pub/scm/linux/kernel/git/akpm/mm.git/commit/?h=mm-unstable&id=bab632d69ee48a106e779b60cc01adfe80a72807.

[103] Steffen Maass et al. "ECOTLB: Eventually Consistent TLBs". In: *ACM Trans. Archit. Code Optim.* 17.4 (Sept. 2020). ISSN: 1544-3566. DOI: 10.1145/3409454. URL: https://doi.org/10.1145/3409454.

[104] Hasan Al Maruf et al. "TPP: Transparent Page Placement for CXL-Enabled Tiered Memory". In: (2022). DOI: 10.48550/ARXIV.2206.02878. URL: https://arxiv.org/abs/2206.02878.

[105] *Message Passing Interface*. URL: https://www.mpi-forum.org/.

[106] Samuel K. Moore. *AMD CEO: The Next Challenge Is Energy Efficiency. A 500-megawatt supercomputer is "probably too much"*. Feb. 2023. URL: https://spectrum.ieee.org/amd-eyes-supercomputer-efficiency-gains.

[107] Djob Mvondo et al. "OFC: An Opportunistic Caching System for FaaS Platforms". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 228–244. ISBN: 9781450383349. DOI: 10.1145/3447786.3456239. URL: https://doi.org/10.1145/3447786.3456239.

[108] Rolf Neugebauer et al. "Understanding PCIe Performance for End Host Networking". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 327–341. ISBN: 9781450355674. DOI: 10.1145/3230543.3230560. URL: https://doi.org/10.1145/3230543.3230560.

[109] *NFS: Network File System Protocol specification*. RFC 1094. Mar. 1989. DOI: 10.17487/RFC1094. URL: https://www.rfc-editor.org/info/rfc1094.

[110] Tu Dinh Ngoc et al. "Mitigating Vulnerability Windows with Hypervisor Transplant". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 162–177. ISBN: 9781450383349. DOI: 10.1145/3447786.3456235. URL: https://doi.org/10.1145/3447786.3456235.

[111] Vlad Nitu et al. "Welcome to Zombieland: Practical and Energy-Efficient Memory Disaggregation in a Datacenter". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190537. URL: https://doi.org/10.1145/3190508.3190537.

[112] *NVM ExpressTM over Fabrics Revision 1.1a*. July 2021. URL: https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1a-2021.07.12-Ratified.pdf.

[113] Takeshi Okuda et al. "A Remote Swap Management Framework in a Virtual Machine Cluster". In: *2010 IEEE 3rd International Conference on Cloud Computing*. 2010, pp. 546–547. DOI: 10.1109/CLOUD.2010.13.

[114] *Open Source Serverless Cloud Platform*. URL: https://openwhisk.apache.org/.

[115] *OpenStack Swift*. URL: https://github.com/openstack/swift.

[116] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. "Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality". In: *Proceedings of the 20th International Middleware Conference Industrial Track*. Middleware '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 1–7. ISBN: 9781450370417. DOI: 10.1145/3366626.3368125. URL: https://doi.org/10.1145/3366626.3368125.

[117] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.

[118] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: https://doi.org/10.1145/361011.361073.

[119] *Proxmox*. URL: https://proxmox.com/en/.

[120] Kashifuddin Qazi and Steven Romero. "Remote Memory Swapping for Virtual Machines in Commercial Infrastructure-as-a-Service". In: *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. 2019, pp. 1–8. DOI: 10.1109/CCCS.2019.8888069.

[121] *QEMU, A generic and open source machine emulator and virtualizer*. URL: https://www.qemu.org/.

[122] Amanda Raybuck et al. "HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 392–407. ISBN: 9781450387095. DOI: 10.1145/3477132.3483550. URL: https://doi.org/10.1145/3477132.3483550.

[123] Charles Reiss et al. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: Association for Computing Machinery, 2012. ISBN: 9781450317610. DOI: 10.1145/2391229.2391236. URL: https://doi.org/10.1145/2391229.2391236.

[124] Cliff Robinson. *NVIDIA H100 Hopper Details at HC34 as it Waits for Next-Gen CPUs*. Aug. 2022. URL: https://www.servethehome.com/nvidia-h100-hopper-details-at-hc34-as-it-waits-for-next-gen-cpus/.

[125] Zhenyuan Ruan et al. "AIFM: High-Performance, Application-Integrated Far Memory". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 315–332. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/ruan.

[126] Adam Ruprecht et al. "VM Live Migration At Scale". In: *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 45–56. ISBN: 9781450355797. DOI: 10.1145/3186411.3186415. URL: https://doi.org/10.1145/3186411.3186415.

[127] Marta Rybczyńska. *Introducing maple trees*. Feb. 2021. URL: https://lwn.net/Articles/845507/.

[128] Solmaz Salehian and Yonghong Yan. "Evaluation of Knight Landing High Bandwidth Memory for HPC Workloads". In: *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*. IA3'17. Denver, CO, USA: Association for Computing Machinery, 2017. ISBN: 9781450351362. DOI: 10.1145/3149704.3149766. URL: https://doi.org/10.1145/3149704.3149766.

[129] Mitsuhisa Sato et al. "Co-Design for A64FX Manycore Processor and "Fugaku"". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2020, pp. 1–15. DOI: 10.1109/SC41405.2020.00051.

[130] Martin Schwidefsky et al. "Collaborative Memory Management in Hosted Linux Environments". In: (Jan. 2006).

[131] Sai Sha et al. "VTMM: Tiered Memory Management for Virtual Machines". In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. Rome, Italy: Association for Computing Machinery, 2023, pp. 283–297. ISBN: 9781450394871. DOI: 10.1145/3552326.3587449. URL: https://doi.org/10.1145/3552326.3587449.

[132] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. "Resource Deflation: A New Approach For Transient Resource Reclamation". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303945. URL: https://doi.org/10.1145/3302424.3303945.

[133] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972.

[134] Kalesh Singh. *Android MGLRU Evaluation*. 2022. URL: https://lpc.events/event/16/contributions/1336/attachments/972/1895/Android%20MGLRU%20Evaluation.pdf.

[135] *Single Root I/O Virtualization and Sharing Specification Revision 1.0*. 2007. URL: https://pcisig.com/single-root-io-virtualization-and-sharing-specification-revision-10.

[136] John D. Slingwine and Paul E. McKenney. *Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring*. 1993. URL: https://patents.google.com/patent/US5442758.

[137] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 1st. Morgan & Claypool Publishers, 2011. ISBN: 1608455645.

[138] Michael Stonebraker. "The Case for Shared Nothing". In: *International Workshop on High-Performance Transaction Systems, HPTS 1985, Asilomar Conference Center, Pacific, Grove, California, 23-25 September 1985*. 1985.

[139] Sayantan Sur et al. "Performance Analysis and Evaluation of Mellanox ConnectX InfiniBand Architecture with Multi-Core Platforms". In: *15th Annual IEEE Symposium on High-Performance Interconnects (HOTI 2007)*. 2007, pp. 125–134. DOI: 10.1109/HOTI.2007.16.

[140] Billy Tallis. *Gen-Z Interconnect Core Specification 1.0 Published*. Feb. 2018. URL: https://www.anandtech.com/show/12431/genz-interconnect-core-specification-10-published.

[141] *The CCIX Consortium*. URL: %https://www.ccixconsortium.com/.

[142] *The Machine: A new kind of computer*. Sept. 2014. URL: https://www.hpl.hp.com/research/systems-research/themachine/.

[143] Kun Tian et al. "CoIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct I/O". In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4.

[144] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL].

[145] Shin-Yeh Tsai and Yiying Zhang. "LITE Kernel RDMA Support for Datacenter Applications". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 306–324. ISBN: 9781450350853. DOI: 10.1145/3132747.3132762. URL: https://doi.org/10.1145/3132747.3132762.

[146] David Hildenbrand & Michael S. Tsirkin. *Virtio-(balloon—pmem—mem): Managing Guest Memory*. 2020. URL: https://static.sched.com/hosted_files/kvmforum2020/8e/KVM%20Forum%202020%20Virtio-%28balloon%20pmem%20mem%29%20Managing%20Guest%20Memory.pdf.

[147] Michael S. Tsirkin and Cornelia Huck. *Virtual I/O Device (VIRTIO) Version 1.1*. Apr. 2019. URL: https://docs.oasis-open.org/virtio/virtio/v1.1/cs01/virtio-v1.1-cs01.html..

[148] *Unified Extensible Firmware Interface (UEFI) Specification*. Mar. 2019. URL: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_final.pdf.

[149] *Vagrant*. URL: https://github.com/hashicorp/vagrant.

[150] Abhishek Verma et al. "Large-Scale Cluster Management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741964. URL: https://doi.org/10.1145/2741948.2741964.

[151] Jerome Vienne et al. "Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems". In: *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*. 2012, pp. 48–55. DOI: 10.1109/HOTI.2012.19.

[152] *virtiofs*. URL: https://virtio-fs.gitlab.io/.

[153] Carl A. Waldspurger. "Memory Resource Management in VMware ESX Server". In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2003), pp. 181–194. ISSN: 0163-5980. DOI: 10.1145/844128.844146. URL: https://doi.org/10.1145/844128.844146.

[154] Chenxi Wang et al. "Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 347–362. ISBN: 9781450367127. DOI: 10.1145/3314221.3314650. URL: https://doi.org/10.1145/3314221.3314650.

[155] Chenxi Wang et al. "Semeru: A Memory-Disaggregated Managed Runtime". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 261–280. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/wang.

[156] Qing Wang et al. "Concordia: Distributed Shared Memory with In-Network Cache Coherence". In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 277–292. ISBN: 978-1-939133-20-5. URL: https://www.usenix.org/conference/fast21/presentation/wang.

[157] Ruihong Wang et al. "The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation". In: *Proc. VLDB Endow.* 16.1 (Sept. 2022), pp. 15–22. ISSN: 2150-8097. DOI: 10.14778/3561261.3561263. URL: https://doi.org/10.14778/3561261.3561263.

[158] Wei Wang and Liang Li. *virtio-balloon: VIRTIO_BALLOON_F_FREE_PAGE_HINT*. 2018. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=86a559787e6f5cf662c081363f64a20cad654195.

[159] Sage A. Weil et al. "Ceph: A Scalable, High-Performance Distributed File System". In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. Seattle, WA: USENIX Association, Nov. 2006. URL: https://www.usenix.org/conference/osdi-06/ceph-scalable-high-performance-distributed-file-system.

[160] B. L. Welch. "The Generalization of 'Student's' Problem when Several Different Population Variances are Involved". In: *Biometrika* 34.1/2 (1947), pp. 28–35. ISSN: 00063444. URL: http://www.jstor.org/stable/2332510 (visited on 08/29/2023).

[161] Matthew Wilcox. *mm: fix XIP fault vs truncate race*. 2015. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=283307c7607de2a06d3bfae4cfbf5a566d457090.

[162] Paul Willmann, Scott Rixner, and Alan L. Cox. "Protection Strategies for Direct Access to Virtualized I/O Devices". In: *USENIX 2008 Annual Technical Conference*. ATC'08. Boston, Massachusetts: USENIX Association, 2008, pp. 15–28.

[163] Gerhard J. Woeginger. "There is no asymptotic PTAS for two-dimensional vector packing". In: *Information Processing Letters* 64.6 (1997), pp. 293–297. ISSN: 0020-0190. DOI: https://doi.org/10.1016/S0020-0190(97)00179-8. URL: https://www.sciencedirect.com/science/article/pii/S0020019097001798.

[164] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. "Towards an Unwritten Contract of Intel Optane SSD". In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotstorage19/presentation/wu-kan.

[165] Wm Wulf and Sally McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *Computer Architecture News* 23 (Jan. 1996).

[166] Zi Yan et al. "Nimble Page Management for Tiered Memory Systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 331–345. ISBN: 9781450362405. DOI: 10.1145/3297858.3304024. URL: https://doi.org/10.1145/3297858.3304024.

[167] Juncheng Yang, Yao Yue, and K. V. Rashmi. "A large scale analysis of hundreds of in-memory cache clusters at Twitter". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 191–208. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/yang.

[168] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.

[169] Efri Zeidner et al. *Internet Small Computer Systems Interface (iSCSI)*. RFC 3720. Apr. 2004. DOI: 10.17487/RFC3720. URL: https://www.rfc-editor.org/info/rfc3720.

[170] Jin Zhang et al. "GiantVM: A Type-II Hypervisor Implementing Many-to-One Virtualization". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 30–44. ISBN: 9781450375542. DOI: 10.1145/3381052.3381324. URL: https://doi.org/10.1145/3381052.3381324.

[171] Yunqi Zhang et al. "History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 755–770. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhang-yunqi.

[172] Han Zhao et al. "Bandwidth and Locality Aware Task-Stealing for Manycore Architectures with Bandwidth-Asymmetric Memory". In: *ACM Trans. Archit. Code Optim.* 15.4 (Dec. 2018). ISSN: 1544-3566. DOI: 10.1145/3291058. URL: https://doi.org/10.1145/3291058.

[173] Junji Zhi, Nilton Bila, and Eyal de Lara. "Oasis: Energy Proportionality with Hybrid Server Consolidation". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: Association for Computing Machinery, 2016. ISBN: 9781450342407. DOI: 10.1145/2901318.2901333. URL: https://doi.org/10.1145/2901318.2901333.

[174] Yang Zhou et al. "Carbink: Fault-Tolerant Far Memory". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 55–71. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/zhou-yang.

[175] Ross Zwisler. *Add support for Heterogeneous Memory Attribute Table*. June 2017. URL: https://lwn.net/Articles/724562/.

Résumé en français

Depuis les années 2000, il est possible de louer des machines physiques (ou serveurs) à distance au sein de centres de données répondant à différents besoins de configuration (ressources, architecture matérielle, ...). La location d'une machine à un unique client, que l'on nomme hébergement dédié, permet de respecter des contraintes d'isolation basées sur la confidentialité, l'intégrité et la disponibilité des applications d'un client. En revanche, l'hébergement dédié répond mal au cas d'utilisation de nombreux clients qui souhaitent utiliser les ressources d'un serveur momentanément ou utiliser une faible quantité de ressources. Hors, chaque ressource inutilisée nécessite un coût énergétique de fonctionnement qui se répercute par des coûts de location élevés. Les centres de données ont proposé un modèle de déploiement alternatif nommé hébergement mutualisé et permettant le partage de ressources physiques entre plusieurs utilisateurs. Afin de permettre l'exécution d'applications de différents clients tout en respectant les contraintes d'isolation, les centres de données ont largement adoptés la virtualisation des ressources physiques avec des unités d'exécution nommés machines virtuelles (VMs). Ainsi, les VMs ont permis une amélioration significative de l'utilisation des ressources d'un centre de données en permettant le partage des ressources.

Malgré l'amélioration significative d'utilisation des ressources offerte par la mutualisation des ressources, les centres de données observent encore un nombre significatif de ressources non utilisées. Deux phénomènes expliquent les raisons pour lesquelles les centres de données peinent à augmenter l'utilisation des ressources avec les machines virtuelles. Premièrement, lorsqu'un utilisateur demande la création d'une VM, un orchestrateur se charge de traduire la configuration désirée en requête d'allocation de ressources en calcul, mémoire, stockage et bande passante réseau sur un des serveurs du centre de données. Mais le problème que doit résoudre l'orchestrateur afin de trouver un placement optimal des VMs est NP-difficile. L'orchestrateur utilise donc des heuristiques de placement qui garantissent une limite maximum non nulle sur les ressources gaspillées au sein du centre de données mais laissent tout de même un ensemble de ressources non allouées. Deuxièmement, à l'issue de l'allocation, les applications s'exécutant au sein de la VM vont progressivement utiliser les ressources allouées lors de la création de la VM. Par construction, les

ressources utilisées par la VM ne peuvent jamais excéder les ressources allouées lors de la création de la VM. Les ressources inutilisées sont la différence entre les ressources allouées et les ressources utilisées à chaque instant. Ces ressources inutilisées constituent également une perte pour les centres de données qui doit les utiliser et préférerait les louer. Par ailleurs, un phénomène nommé sur-provisionnement participe à accentuer la quantité de ressources inutilisées. En effet, l'exécution d'une application est ralentie si une application nécessite davantage de ressources que la limite imposée lors de l'allocation. Afin d'éviter ces ralentissements, les utilisateurs cherchent à sur-provisionner les VMs lors de l'allocation ce qui résulte en une augmentation des ressources inutilisées si les applications ont des phases de repos.

Notre première proposition cherche à améliorer l'utilisation des ressources non allouées sur les serveurs par l'orchestrateur. Notre prototype permet à des applications s'exécutant dans une VM d'effectuer des accès à des ressources mémoire distante de manière transparente. Nous propos de reposer sur l'algorithme de remplacement de page mémoire du noyau Linux et d'y intégrer le support pour la réservation à grain fin des ressources mémoires. Le prototype réduit l'impact sur les performances des applications invitées en utilisant le réseau RDMA pour les communications, et en s'intégrant avec la gestion mémoire du système d'exploitation de la VM.

Notre seconde proposition cherche à améliorer l'utilisation des ressources inutilisée sur un seul serveur. Ainsi, nous présentons les résultats de l'analyse de l'ensemble des techniques existantes qui essaient d'ajuster la capacité mémoire d'une VM par rapport à la mémoire désirée par les applications. Notre travail montre notamment que les solutions actuelles implémentées au niveau de l'hyperviseur sont trop lentes pour supporter le partage des ressources mémoire et conduisent à des ralentissements ou interruptions de service des VMs. En utilisant les informations issues de notre analyse, nous présentons un prototype permettant d'adapter rapidement la capacité mémoire d'une VM. Pour ce faire, nous proposons de déléguer au système invité la responsabilité de changer sa propre capacité mémoire ainsi que plusieurs optimisations sur les mécanismes permettant des changements de capacités mémoires.

ECOLE
DOCTORALE

Titre : Amélioration de l'utilisation de ressources mémoires pour les machines virtuelles

Mots clés : Hyperviseurs, RDMA, Système d'exploitation, Mémoire hétérogène

Résumé : Les centres de données reposent largement sur les machines virtuelles comme unité de déploiement afin de garantir une isolation forte entre deux déploiements. L'introduction de la virtualisation a permis une amélioration significative de l'utilisation des ressources d'un datacenter par rapport à la location d'une machine physique pour chaque utilisateur. Malgré ce gain, les machines virtuelles offrent toujours un gain d'utilisation de ressources plus faible que l'isolation fournie par le système d'exploitation à travers des processus. Deux phénomènes expliquent le gain relatif des machines virtuelles : au moment de l'allocation d'une VM, un orchestrateur traduit l'allocation en une requête d'allocation de plusieurs ressources sur plusieurs serveurs. Le problème que doit résoudre l'orchestrateur afin de trouver un placement optimal des VMs est NP-difficile, imposant à l'orchestrateur l'utilisation d'heuristiques de placement qui laissent une part de ressource inutilisées sur chaque serveur. De plus, durant l'exécution de la VM, la consommation mémoire augmente à mesure que les accès sont effectués et la différence entre allocation mémoire et utilisation mémoire résulte en un ensemble de ressource mémoire qui ne peuvent être utilisé sans risque de crash de VMs. Dans un premier temps, nous proposons un prototype permettant d'accéder aux ressources mémoires inutilisées sur d'autres serveurs. Notre solution est transparente pour les applications s'exécutant dans une VM et offre une réservation à grain fin des ressources mémoires distantes. Dans un second temps, nous présentons les résultats de notre étude des techniques existantes permettant d'ajuster la capacité mémoire d'une VM sur la mémoire qu'elle utilise. Notre travail montre que les solutions actuelles implémentées au niveau de l'hyperviseur introduisent une dégradation des performances des VMs ainsi que des temps de réponse élevés empêchant le partage des ressources mémoires par plusieurs VMs. Nous proposons une solution utilisant les informations disponibles dans la VM pour adapter rapidement leurs capacités mémoires.

Title : Improving memory usage in virtual machines

Keywords : Hypervisors, RDMA, Operating System, Heterogeneous Memory

Abstract : Data-centers rely on virtual machines (VMs) to offer isolation between deployments. While, the use of VMs enables better resource usage compared to running a service per bare-metal machine, it achieves poorer resource usage than multi-processes solutions. This is caused by two phenomenon : At VM allocation time, VMs are scheduled as resource requests on a VM scheduler which perform virtual machine allocations across a set of servers. Optimal solution to this scheduling problem is NP-hard leading to the adoption of heuristic based allocation that let a good percentage of unallocated memory on each servers known as 'stranded memory'. At VM runtime, VM memory is consumed on-demand and the difference between memory allocation and usage results in a decent portion of 'allocated unused memory' currently impractically usable.
First, we propose a transparent solution for applications running inside VMs to remotely access stranded memory in remote machines with fine-grained reservation of remote resources. Second, we review current techniques trying to fit allocated memory to used memory. We show that all these techniques are managed by the hypervisor and introduce performance degradation in VMs and more importantly high response time which makes resource sharing unpractical. Instead, we propose an abstraction to perform VM-initiated memory provisioning and we present early result of fast adaptation of VM memory.

Institut Polytechnique de Paris
91120 Palaiseau, France