Novel memory and I/O virtualization techniques for next generation data-centers based on disaggregated hardware

Maciej Bielski

▶ **To cite this version:**

Maciej Bielski. Novel memory and I/O virtualization techniques for next generation data-centers based on disaggregated hardware. Hardware Architecture [cs.AR]. Université Paris Saclay (COmUE), 2019. English. ⟨NNT : 2019SACLT022⟩. ⟨tel-02464021⟩

HAL Id: tel-02464021

https://pastel.hal.science/tel-02464021v1

Submitted on 2 Feb 2020

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

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

HAL Authorization

université
PARIS-SACLAY

TELECOM ParisTech

NNT : 2019SACLT022

Thèse de doctorat

Nouvelles techniques de virtualisation de la mémoire et des entrées-sorties vers les périphériques pour les prochaines générations de centres de traitement de données basés sur des équipements répartis déstructurés

Thèse de doctorat de l'Université Paris-Saclay
préparée à Télécom ParisTech

Ecole doctorale n°580 Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Biot Sophia Antipolis, le 18/03/2019, par

M. MACIEJ BIELSKI

Composition du Jury :

Frédéric Pétrot Professeur, Université Grenoble Alpes, ENSIMAG, TIMA/SLS	Rapporteur
Alain Tchana Professeur, Université Côte d'Azur, CNRS/I3S	Rapporteur
Vania Marangozova-Martin Maîtresse de conférences, Université Grenoble Alpes, LIG	Examinatrice
Melek Önen Maîtresse de conférences, EURECOM	Examinatrice
Laurent Pautet Professeur, Telecom ParisTech	Examinateur
Alvise Rigo Ingénieur R&D, Virtual Open Systems	Examinateur
Guillaume Urvoy-Keller Professeur, Université Côte d'Azur, CNRS/I3S	Examinateur
Renaud Pacalet Directeur d'études, Telecom ParisTech	Directeur de thèse

Acknowledgments

The work presented in this dissertation was carried out by the author when he was an employee of Virtual Open Systems. This work was supported by the *dReDBox* project. This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 687632. This work reflects only authors' view and the EC is not responsible for any use that may be made of the information it contains.

At the end of my doctorate I would like to acknowledge everyone who helped me to reach this stage of my education track.

I would like to thank teachers and professors who introduced me to science and engineering and somehow made me interested in exploring them on my own. In particular, I would like to thank Professor Renaud Pacalet for supervising my PhD and providing substantial help in improving the thesis manuscript. Several improvements was also suggested by Professor Frédéric Pétrot and Professor Alain Tchana, who spent their time on reviewing the manuscript for which I am immensely grateful. I would also like to thank fellow members of LabSoC, including Professor Ludovic Apvrille who actually brought me to the laboratory where I was making my first steps in hacking Linux kernel during my internship.

While working on my PhD I was employed by Virtual Open Systems and therefore I would like to thank Daniel Raho, CEO of the company, for offering me this opportunity. Moreover, I thank all colleagues I worked with, especially Alvise Rigo, who was my company supervisor for most of the time and helped me to solve many technical challenges.

Last but not least, my family and my friends deserve big words of thank for keeping me happy everyday. In particular, I would like to thank my parents who substantially supported my education from the very beginning. Although it required an effort and determination from myself, all of that could go for nothing without a solid foundation they provided.

Finally, I would like to especially thank my beloved Magda, a hero in the shadows who was accompanying me for all this time with a profound understanding and made me enjoy life more than ever before.

Contents

List of Figures 7

List of Tables 9

List of Terms 11

List of Publications 13

1 Introduction 15

1.1 Data centers — current state 17
1.2 Clustered architecture 21
1.3 Virtualization role 22
1.4 Clustering drawbacks 24
1.5 Disaggregated architecture 28
1.6 Disaggregated systems and virtualization 31
1.7 Focus and scope of this work 32

2 Related work 35

2.1 Memory disaggregation 35
2.2 VM memory provisioning and balancing 47
2.3 Uniform address space 53
2.4 Inter-VM memory sharing and migration 54
2.5 Devices disaggregation 58

3 Guest memory provisioning in a disaggregated system 65

3.1 Chapter introduction 65

3.2 Proposed system architecture 66
3.3 Resize volume 70
3.4 Live VM balancing: guest parameters visibility 72
3.5 Explicit resize requests 73
3.6 Request path 75
3.7 Resize granularity 75
3.8 Disaggregation context 76
3.9 Guest memory isolation 76
3.10 Chapter conclusion 77

4 VM memory sharing and migration **79**
4.1 Chapter introduction 79
4.2 Memory sharing — overview 80
4.3 VM migration — overview 81
4.4 Software modifications 82
4.5 Proposed system architecture 83
4.6 Sharing disaggregated memory 83
4.7 VM migration 96
4.8 Chapter conclusion 101

5 Disaggregated peripherals attachment **103**
5.1 Chapter introduction 103
5.2 Device emulation and direct attachment 104
5.3 Disaggregated passthrough design 108
5.4 Chapter conclusion 108

6 Implementation and evaluation **109**
6.1 Memory provisioning 109

7 Conclusion **119**
7.1 Perspectives and future works 121

Appendices **131**

A Disaggregated peripherals attachment - confidential part 133

A.1 Disaggregated passthrough design 134

A.2 Infrastructure setup in more details 140

A.3 IOMMU maps update on guest RAM resize 145

A.4 Disaggregated device detachment 146

List of Figures

1-1 Clustered system architecture 22
1-2 Disaggregated system architecture 28

2-1 Categorization of works related to memory disaggregation 36
2-2 Comparison of runtime guest memory resizing methods 47
2-3 Simple illustration of a Uniform Address Space concept 53
2-4 Hardware-based remote device sharing methods 58

3-1 Guest memory provisioning from isolated pool 66
3-2 Dynamic memory resize 71
3-3 Different moments of issuing memory resize request 73

4-1 Virtualization framework components 84
4-2 Sharing disaggregated memory 84
4-3 Mapping a disaggregated *arbitration unit* located at a memory node . . 90
4-4 Disaggregated shared memory attachment 93
4-5 VMs of the same sharing group reusing dedicated slots 95
4-6 Dedicated slot initialization from an existing RAM 96
4-7 Two meanings of a disaggregated VM migration 97
4-8 Interconnect reconfiguration steps 100

5-1 Direct attachment compared to device emulation 104
5-2 Standard passthrough not possible on disaggregated system 105

6-1 System prototype components 110
6-2 Latency results from `Redis-benchmark` against the *SimpleDB* application 117

A-1 Disaggregated passthrough — architecture design 135
A-2 Direct attachment initialization 137
A-3 Device attachment configuration at GSO side 141
A-4 Updating directly attached device after guest memory resize 145
A-5 Updating directly attached device after — GSO part 147

List of Tables

2.1 Inter-VM memory sharing methods 55

6.1 Latencies of memory resize steps 113

List of Terms

AU Arbitration Unit, a module granting an access right to a region of memory shared between multiple virtual machines using it in a concurrent way.

BAR Base Address Register

CPU Central Processing Unit

DRAM Dynamic random-access memory

GPA Guest Physical ADdress space, an address space created by a virtual machine and exposed to the guest OS, which will consider it as if it was a HPA.

GSO Global System Orchestrator, a central (at least logically) management unit of a data-center handling configuration of all nodes and the interconnect to keep the system working. Holds the registry of all deployed VMs and their configurations at any given moment. Physically can be implemented as a single machine or in a distributed way.

GVA Guest Virtual Address space, same as HVA but in the guest OS.

HPA Host Physical Address space, used to describe a physical layout of resources available at each system node. Different ranges correspond to registers of different physical resources. This is a base addressing scheme determined during hardware production time.

HVA Host Virtual Address space, maintained by an operating system by building translation maps (called *page tables*), usually traversed by the *memory management*

unit (MMU) for better performance. In fact, some operating system may not use virtual addresses, especially small embedded ones.

IC Interrupt Controller

IOMMU Input-output memory management unit, connects DMA-capable devices to the main memory, provides address translation between a device address space and the host physical address space and filters accesses to the memory at the same time.

MMU Memory Management Unit

MPI Message Passing Interface

NVM Non-volatile memory

RDMA Remote Direct Memory Address

TCO Total Cost of Ownership

TLB Translation Lookaside Buffer, buffer caching recent virtual-to-physical address mappings translated by the MMU.

UPA Uniform Physical ADdress space, a very large address space mapping a HPA of each system node as a distinctive range.

VM Virtual Machine, a software abstraction of a computer system hardware, optionally supported by hardware extensions in order to accelerate certain operations.

List of publications

Conference papers:

- M. Bielski, C. Pinto, D. Raho, R. Pacalet, *Survey on memory and devices disaggregation solutions for HPC systems*, 19th International Conference on Computational Science and Engineering (CSE 2016)
- D. Syrivelis, A. Reale, K. Katrinis, I. Syrigos, M. Bielski, D. Theodoropoulos, D. N. Pnevmatikatos and G. Zervas, *A Software-defined Architecture and Prototype for Disaggregated Memory Rack Scale Systems*, International Conference on Embedded Computer Systems: Architectures, Modelling, and Simulation, (SAMOS XVII 2017)
- D. Theodoropoulos, A. Reale, D. Syrivelis, M. Bielski, N. Alachiotis, D. Pnevmatikatos, *REMAP: Remote mEmory Manager for disAggregated Platforms*, 29th International Conference on Application-specific Systems, Architectures and Processors, (ASAP 2018)
- M. Bielski, I. Syrigos, K. Katrinis, D. Syrivelis, A. Reale, D. Theodoropoulos, N. Alachiotis, D. Pnevmatikatos, G. Zervas, V. Mishra, A. Saljoghei, A. Rigo, M. Enrico, V. Mishra, A. Saljoghei, E. Pap, , O. González de Dios, Dionisios N, Pnevmatikatos, J. Fernando Zazo, S. Lopez-Buedo, M. Torrents, F. Zyulkyarov, *dReDBox: Materializing a Full-stack Rack-scale System Prototype of a Next-Generation Disaggregated Datacenter*, prototype demonstration and publication in Design, Automation and Test in Europe, (DATE 2018)
- M. Enrico, V. Mishra, A. Saljoghei, M. Bielski, E. Pap, I. Syrigos, O. González de Dios, D. Theodoropoulos, D. N Pnevmatikatos, A. Reale, D. Syrivelis, G. Zervas, N. Parsons, K. Katrinis, *Demonstration of NFV for Mobile Edge Computing on an Optically Disaggregated Datacentre in a Box*, Optical Fiber Communication Conference, (OFC 2018)
- A. Saljoghei, V. Mishra, M. Bielski, I. Syrigos, K. Katrinis, D. Syrivelis, A. Reale, D. N. Pnevmatikatos, D. Theodoropoulos, M. Enrico, N. Parsons, G. Zervas,

dReDBox: Demonstrating Disaggregated Memory in an Optical Data Centre, Optical Fiber Communication Conference, (OFC 2018)

- M. Bielski, A. Rigo, R. Pacalet, *Dynamic guest memory resizing for disaggregated systems – paravirtualized approach*, 27th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2019) [accepted]

Patents:

- M. Bielski, A. Rigo, M. Paolino, D. Raho, *Disaggregated computing architecture*, Owner:Virtual Open Systems, (proposal submitted)

Open source contributions:

- M. Bielski, A. Reale, *Memory hotplug support for arm64*, Linux kernel patch, link: https://lkml.org/lkml/2017/11/23/182

Chapter 1

Introduction

This dissertation is positioned in the context of the system disaggregation - a novel architectural approach expected to gain popularity in the data center sector. In traditional clustered systems resources are provided by one or multiple machines characterized by a fixed amount of memory, CPU cores and set of available devices. Differently to that, in disaggregated systems resources are provided by discrete nodes, with each node providing only one type of resources. Instead of a machine, the term of a slot is used to describe a workload deployment unit. The crucial difference is that a slot does not provide a fixed amount of resources but it is dynamically assembled before a workload deployment by the unit called system orchestrator.

In the following sections of this chapter, we present the drawbacks of the clustering approach and motivate the architecture disaggregation. Furthermore, we add the virtualization layer to the picture as it is a crucial part of data center systems nowadays and it is expected to retain its position also in disaggregated ones. That is because virtualization provides an isolation between deployed workloads and a flexible resources partitioning. However, it needs to be adapted in order to take full advantage of disaggregation. Thus, the main contributions of this work are focused on the virtualization layer support for disaggregated memory and devices provisioning.

In the chapter 2 we present the state of the art analysis related to the subject of this dissertation. The analysis is divided into few sections, according to specific aspects of disaggregation like *Virtual Machine* (VM) memory provisioning, inter-VM memory sharing and remote device attachment.

Our first main contribution is described in Chapter 3. It presents the software stack modifications related to VM memory provisioning, which allow to adjust the amount of guest (running in a VM) RAM at runtime on a memory section granularity. From the software perspective it is transparent whether they come from local or remote memory banks.

In Chapter 4 we extend the proposals of Chapter 3 to allow inter-VM memory sharing and VM migration on a disaggregated architecture. That is the second main contribution of this dissertation. First, we present how regions of disaggregated memory can be shared between VMs running on different nodes. This sharing is designed in a way that involved guests are oblivious to the fact of being co-located on the same computing node or not. Additionally, we discuss different flavors of concurrent accesses serialization methods. We then explain how the VM migration term gained a twofold meaning. Because of resources disaggregation, a workload is associated to at least one computing node and one memory node. It is therefore possible that it is migrated to a different computing node and keeps using the same memory, or the opposite. We discuss both cases and describe how this can open new opportunities for server consolidation.

Chapter 5 provides our last main contribution related to disaggregated peripherals virtualization. Starting from the assumption that the architecture disaggregation brings many positive effects in general, we explain why it breaks the passthrough peripheral attachment technique (also known as a direct attachment), which is very popular for its near-native performance. To address this limitation we present a design that adapts the passthrough attachment concept to the architecture disaggregation. By this novel design, disaggregated devices can be directly attached to VMs, as if they were plugged locally. Moreover, all modifications do not involve the guest OS itself, for which the setup of the underlying infrastructure is not visible.

Chapter 6 presents a prototype that we used to evaluate the implementation of our ideas and provides preliminary results about flexible guest memory resizing.

Finally, Chapter 7 concludes the dissertation and proposes several directions of future works.

1.1 Data centers — current state

Perhaps the first image, derived from news media or movie scenes, that one associates with the term **data center** is a very large facility filled with multiple server cabinets and connected by hundreds of meters of cables. Typically thousands of blinking diodes and a constant noise produced by air-conditioning are the only indicators that there is actually something going on there. Such a description is only roughly accurate. Many data centers are indeed occupying a surface from hundreds up to millions of square meters and hosting huge and powerful installations called sometimes supercomputers (see the TOP500[8] list, ranking the most powerful computers in the world). But most of them are much smaller, for example private enterprise installations may occupy just one or few rooms but still provide enough resources to serve well a small- and medium-sized company or institution.

As of 2018, when talking about a data center we consider an installation hosting at least a dozen of *Central Processing Unit* (CPU) cores together with available memory in the order of at least several dozens of gigabytes. A crucial role in the system is also played by installed accelerators, which are off-loading CPUs from certain types of computations. Definitely, the volume of resources is much bigger than what is offered by personal computers or mobile devices nowadays.

As a consequence of the desired performance level and the number of components, these systems typically require significant investments in order to be deployed and consume a big amount of electrical energy. This is commonly described by the *Total Cost of Ownership* (TCO) term, which is one of the most important factors determining whether a given data-center technology will be successful or not. A system can also be characterized by *performance-per-dollar*, a ratio between the offered operational capabilities and the cost.

Not surprisingly, such large-sized installations are meant to be shared by multiple workloads (or users) in order to maximize the utilization factor (and an associated profitability factor) and minimize energy consumption (and related heat production) per workload, while still keeping the offered performance aligned with modern applications requirements. Resource partitioning is usually done by virtualization techniques, which allow to abstract underlying physical resources so that they are provided in the form of a

Virtual Machine (VM), which is a workload deployment vehicle and, from a workload's perspective, creates an impression that it uses the system exclusively.

Trends

The fact of exporting both computations and data to an external data center is commonly known as *cloud computing*. Services offered by cloud system providers are usually falling into one of three basic models, namely *Infrastructure-/Platform-/System-as-a-Service* (*IaaS*, *PaaS* and *SaaS*, respectively) [3]. Without going into details and differences between them, which are not important here, a common incentive for users is to focus on their particular mission while moving the responsibility of computing resources provisioning to an external provider [34]. Additionally, it can be a way to optimize the operational costs since the subscription can usually be flexibly changed according to customer requirements at a given moment, instead of purchasing physical industry-grade and expensive hardware to fulfill the peak-usage scenario. Subscription fees, similarly to other goods available on the market, consist of two parts. The first one is governed by purely business factors like product popularity or market competition level — these are out of interest in this work. The second one, though, is determined by technical factors, like an amount of resources reserved for a given workload and therefore unavailable for other system users.

A very important aspect is that not all reserved resources may be effectively used by the customer. For example several machines may be necessary to provide the required amount of memory resources, but not all of associated CPUs may be needed. In consequence, the corresponding computational power will be wasted because, in a clustered architecture, there is no way to attach unused CPUs to other machines. This is clearly an inefficiency which a customer needs to pay for.

Important examples of data-center usage are computations typically categorized as *Big Data*, processing large amounts of data from various sources and for different analytics purposes. Tasks of this type are very often leveraging distributed/parallel processing algorithms and thus require significant processing power as well as storage for input and output data. For such use-cases the *cloud computing* model fits very well.

A recent literature study [34] provides real-world examples that help to create an image

of different computation volumes (these are numbers from 2014, they can be even higher in 2018). The Large Hadron Collider at CERN is expected to produce around 15 petabytes of data every year. A Boeing jet engine can generate up to 20 terabytes of data during one hour of operation. Facebook servers are processing about 500 terabytes of user log data and hundreds of terabytes of image data on a daily basis. Similarly, servers used by eBay are achieving a hundred of petabytes processed within the same period of time.

Browser requests, social media applications, customers profiling algorithms, personal health care monitoring, blockchain transactions or manufacturing chains — every day we are surrounded by myriad of situations in which data is produced, collected and analyzed. With the upcoming era of 5G communication and IoT (*Internet of Things*) this tendencies will likely get reinforced.

Global impact

It has been estimated that the global data traffic grew 56 times from 2002 until 2009, while the computational power increased 16 times in the same period. By extrapolation, it was expected to take around 13 years for a 1000-fold increase in computational power — theoretically (as of 2014). However, hardware energy efficiency is not expected to grow more than 25-fold at the same time. Such divergence of trends indicates that data centers growth may imply a significant increase in a global energy consumption or non-sufficient energy supply may become an actual limit for the development of the sector [34]. Another publication reported that US data centers contributed to 2.2% of the country total electricity consumption in 2013, corresponding to 100 million metric tons of carbon pollution [44]. It was also predicted that between 2012 and 2020 the carbon footprint of the sector will globally grow by 7% year-to-year, which makes it the fastest growing ICT sector in this regard [49].

A conclusion that one may draw from the presented numbers is that there is definitely an urgent need for limiting the amount of energy consumed by data centers worldwide. One way to achieve this would be decreasing the number of data centers but the feasibility of it is questionable, to say the least. Another approach consists in improving the energy efficiency of data centers, with a two-fold meaning. Firstly, by making

them consume less energy for the same given amount of work. Secondly, by reducing the amount of wasted energy, that is, situations where some resources remain powered while they are not efficiently used. The latter is indeed reported as the easiest way to lower the total energy expenditure [60]. A characteristic associated to energy efficiency is the *server consolidation* term. It is not an absolute metric but it describes the ability of a server to execute a maximum amount of computations on a minimal amount of resources.

The need for more energy efficient data centers was already postulated in the literature and supported with real-world examples [27]. On a Twitter production cluster, for instance, it was measured that the aggregated CPU utilization is constantly lower than 20%, while the reservation is between 60% and 80%. This means that 40% to 60% of CPUs remain ready to perform computations but stay idle. With regard to memory utilization, between 40% and 50% is actively used, while around 80% is reserved, therefore 30% to 40% of the memory resource is wasted. In Google clusters, another example from the top-tier operators, it was observed that typically from 25% to 35% of CPUs are actively used, while 75% are reserved. This gives from 40% to 50% of computing power loss. And 40% of the memory resource is actively used, while 60% in average is reserved. Still, in both cases there is no information about non-reserved resources, the authors do not specify whether they are consuming any energy or not.

In addition, the authors explain that the above numbers are representing only a small group gathering the biggest market players, accounting for 5% to 7% of the total number of installed servers. These major players can probably afford the adoption of standard and/or custom solutions in order to improve the energy efficiency of their servers. They might even be forced to do so because, for them, a limited energy supply at a given location may otherwise become a scalability issue. Other small- and medium-sized data centers have neither such incentives nor comparable resources. Moreover, they often stay conservative in terms of technology upgrades because of the risks related to modifying an already working infrastructure, even if it is sub-optimal [60]. Therefore, for the majority of the global data-center market, the energy effectiveness is probably even worse than the numbers presented above.

Another frequently mentioned problem is the correct estimation of resources. Ac-

cording to one study, 70% of workloads overestimate their required resources up to 10 times [27]. Exact numbers would vary for each workload but the most important message is the high level picture. It shows that workloads are either profiled with low accuracy or a substantial estimation margin is introduced by purpose to avoid potential performance issues.

It was suggested that, in a global perspective, it would be more efficient if the workloads of multiple medium- and small- installations could be aggregated and deployed together on bigger systems offered by multi-tenant providers [30]. Such data centers are expected to offer a logical slice of resources to rent, according to customer needs. However, under the hood all workloads would run on the same physical hardware taking advantage of efficient balancing techniques for better utilization level. Moreover, such providers, which business is focused on infrastructure provisioning, are also supposed to enhance their technology more frequently in order to keep the energy efficiency level as high as possible. Especially, they could adopt architecture disaggregation, which underlies our work.

Regardless of whether one would consider only the financial aspect or also the environmental impact, the problem of low energy efficiency of the data centers sector should definitely not be neglected. In the following we discuss how the situation can be improved by switching to a novel, disaggregated hardware architecture.

1.2 Clustered architecture

At present, the data-center sector is dominated by cluster systems that achieve their great capabilities by combining together multiple smaller machines, as illustrated on figure 1-1. Each machine, also referred by the term of a slot is built in a traditional way, in the sense of being composed of CPUs and operating memory banks, both physically bound to the same motherboard [28]. In addition, different types of devices attached may also be attached to the slot (e.g. hard disks, network interface cards, GPUs or accelerators).

The whole system is managed by the *Global System Orchestrator* (GSO). Different implementations may use different naming for this module but its purpose stays the same.

Amongst others, it decides on which machine a given workload will be deployed (load balancing), establishes intra-cluster connectivity, monitors various system parameters or manages machine upbringing and shutdown. This is also the unit which usually provides an administrative interface for system configuration.
The interconnect provides a high-speed connectivity between all machines so that they are capable of passing messages as well as exchange data. Typically, this is used by distributed programming frameworks (like OpenMPI, an open-source implementation of the MPI standard [19]) in order to partition a computational problem into smaller chunks that can be processed in parallel on multiple machines with proper synchronization.

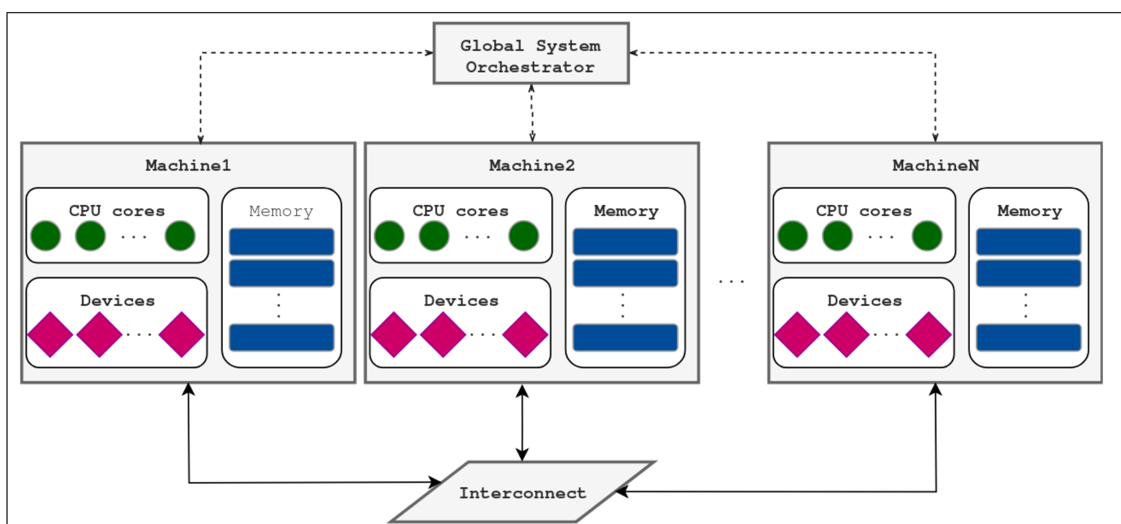


Figure 1-1: Clustered system architecture

1.3 Virtualization role

An essential part of nowadays data-center systems is the virtualization layer, provided by the software stack component called *hypervisor*. The hypervisor is tightly coupled with an operating system kernel and it is responsible for management of VMs. Virtualization benefits the system in several ways:

1. Virtualization abstracts the underlying physical resources and makes (usually a slice of) them accessible to the user in a convenient way — as a logical unit called *Virtual Machine* (VM). A user needs to know neither the physical system

architecture nor the actual location of the components. According to a variable system load a VM could be migrated to a different slot and such operation, ideally, should be barely noticeable from the workload perspective (in practice, a temporary performance drop is possible). Without the virtualization layer each workload would either need to support migration in its own specific way or it would simply need to be terminated and spawned on a different slot.

2. Physical resources partitioning performed by the hypervisor allows to allocate them in an optimal way to perform a maximum amount of computations on a given amount of resources. With respect to the utilization factor, it is much more efficient than the bare-metal deployment of each workload on a separate machine. Such allocation flexibility applies to both an initial VM setup as well as to runtime resource resizing. The situation is the most straight-forward in the case of device sharing: an accelerator can be attached by a VM either periodically or all the time but in a shareable manner. Regarding the amount of CPU cores used by a VM, it can be changed as long as there are enough resources available on a given machine. Concerning the amount of RAM attached to a VM, a very negative phenomenon is the memory overprovisioning. That is when the amount of RAM reserved for a VM exceeds the needs of a workload for most of its execution time and only during rare peak usage moments all memory resources are effectively used. It is the role of a virtualization layer to support the dynamic resize of VM memory volume in order to limit memory overprovisioning (although the situation is still far from good, as mentioned in section 1.1).

3. Multiple VMs are running independent workload-specific software stacks, including their own *guest operating systems*, which provides a strong isolation between VMs hosted by the same data-center. This feature is a great advantage weighted in favor of virtualization in terms of system security. For example, a critical error within an application executed in one VM may cause a system crash but it will not affect other workloads running in different VMs. The latter could be the case without virtualization, if all workloads were executed as a separate processes of the same operating system.

4. Regarding the guest OS, a strongly desired characteristic provided by virtualization is the ability of VMs to host standard operating systems, optionally equipped with additional installable software modules. From a potential user perspective this greatly facilitates the migration from a workstation to the cloud as the original working environment can stay almost unmodified and the compatibility between an OS and customer's software is not threatened.

Eventually, having enumerated multiple reasons for virtualization adoption, it is critically important that this convenient abstraction does not incur significant performance overhead. The key point here is that the heaviest operations performed by the hypervisor are accelerated by the underlying hardware (e.g. nested paging [7]).

1.4 Clustering drawbacks

In a clustered system, resource provisioning is performed on a slot basis. While slots may be divided into groups of different resource proportions, these proportions are fixed and therefore a given amount of workloads deployed on a given slot may consume completely one type of resources while leaving the others underutilized.
For example, the memory of a slot, which CPUs are powered-off, cannot be easily shared with other slots. Although distributed algorithms are capable of involving multiple machines providing different resources, all worker slots need to communicate with a root slot in order to synchronize the execution or transfer data. Even if one machine in a cluster has a certain amount of memory available, it still needs at least one CPU core in addition, to execute the synchronization code.
Therefore, the crucial problem of clustering is that underutilized resources of one slot cannot be easily attached to another one while they remain powered-on. This leads to an inefficient utilization at a single machine level and further magnified at scale as typically data center installations are composed of numerous slots. The powered-on and non-used resources contribute to the system total energy footprint as well as they produce additional heating — which boils down to yet additional chunk of energy spent in order to dissipate the heat. All in all, clustered systems are marked by the *energy proportionality* issue, in the sense that the total system power consumption is

not proportional to the load. Instead, the power consumption is determined by the number of machines that have to be kept active in order to meet the load requirements. Assuming that a fraction of resources within each machine is likely underutilized, the sum of associated power consumed by all such fractions yields the disproportion [45].

Additionally to allocation inefficiency, recent observations and predictions suggest that the way how resources are provided in clustered architectures does not fit requirements of current workloads. The number of CPU cores per socket is expected to grow up to two times every two years. At the same time, the observed number of VMs per core is also increasing, as is the memory footprint per VM. This means that there is a large demand for bigger and bigger amount of memory to be available per CPU socket. In contrast, the growth of the memory capacity per socket is projected to be much slower and it is stated that an inadequate memory supply may become a scalability limitation (which phenomenon is referred as the *memory capacity wall*). Together with this observation there is postulated a need for new architectural approaches that would allow to expand the available memory independently from the computing resources [39].

Currently, the recommended amount of supplied resources differs depending on the performance benchmarking suite used (High Performance Linpack [6], used to rank the TOP500 list [8] vs. High Performance Conjugate Gradients [4]), however it has been estimated that a modern system should offer at least 0.5GB of main memory per core for the optimal system performance [65]. For example, the first machine on the TOP500 list as of June 2018 is the IBM Summit with a ratio of 1.23 GB of RAM per core [18].

Going further, clustering suffers from yet another drawback related to the availability of accelerators, especially of specialized ones with a relatively high price tag. They are used to accelerate particular types of computing operations (e.g. GPUs for vector or matrix operations), and thus to save CPU clocks, but it is usually not required to have them attached permanently to all machines; many applications might not be able to make use of them. Therefore, it has been noticed that it is in general good to have few of them globally accessible to accelerate some workloads [30].

Efficient peripherals sharing between different slots in a cluster is not straightforward. When a workload needs to use a peripheral which is not available from its current machine, and assuming it runs in a virtualized environment, it must be migrated. This

incurs a significant operation downtime because of the volume of data to be transferred (typically several gigabytes). Additionally, a limited availability of the powerful accelerators may constrain the slot selection algorithms in such a way that the selection will become sub-optimal with regard to the overall server consolidation level. For example, in order to be able to use an accelerator, a workload may have to be deployed on a machine that could otherwise remain powered-off. But once deployed it may not effectively use all the available computing power or RAM, which, in such a case, will be wasted.

Finally, the machine-centered architecture is characterized by a low flexibility of modifications at hardware components level. As the memory banks and CPUs of a machine are physically attached to a common motherboard, the possibilities of upgrading independently only one of those resources may be limited, if not impossible, because of hardware compatibility constraints. This may even lead to *vendor lock-in* situations and impose significantly higher costs for such hardware upgrades, even in the case of commodity systems. This actually limits market competition, which is another negative aspect.

The aforementioned inefficiencies and limitations stem from the presented system architecture. In order to thoroughly address them, there is a need for a novel solution bringing a transparent resource allocation across machines, breaking the fixed proportionality limitation and therefore redefining the notion of a slot [39]. It has been proposed to significantly reshape the hardware topology and improve the overall system heterogeneity such that different types of resources may be reserved separately, according to a workload profile, regardless of the motherboard they are plugged into. This new approach is called *disaggregation* and opens for an improved consolidation level of data-center systems, an increased resources utilization, improved energy proportionality and lower maintenance costs. This is because the new approach assumes that what is currently considered a slot will no longer mean a physical machine of fixed resource proportions but a dynamically assembled instance with different types of resources provided according to workload requirements.

Furthermore, in order to stamp a meaningful mark, such technological shift is expected from commodity-grade products used by small- and medium-sized as well as corporate

data centers, which represent the majority of the sector players (for example, in 2011 their market share in U.S. was estimated to 70% in terms of number of servers and 76% with respect to electricity consumption [60]). Otherwise, a heavily customized technology would be affordable only to a small number of top-tier providers and therefore it wouldn't have a chance to significantly change the situation in a global perspective.

1.5 Disaggregated architecture

Motivation

As discussed in the previous chapter, the data center sector is currently facing several problems which induce both economical and environmental effects on a global scale. To wit, an accurate estimation of the required workload resources is very difficult and they are usually over-provided to satisfy the worst-case scenario [47]. Furthermore, resource sharing flexibility is limited and for that reason a certain fraction of them is not used, except for burst periods of maximum demand. On top of that, within a single machine different types of resources are attached to a common motherboard. Therefore, underutilized resources still remain powered-on and the associated chunk of consumed energy is wasted, which is an *energy proportionality* issue.
In order to address these problems, a new *disaggregated architecture* has been proposed which structure and characteristics are the subject of this chapter.

Design

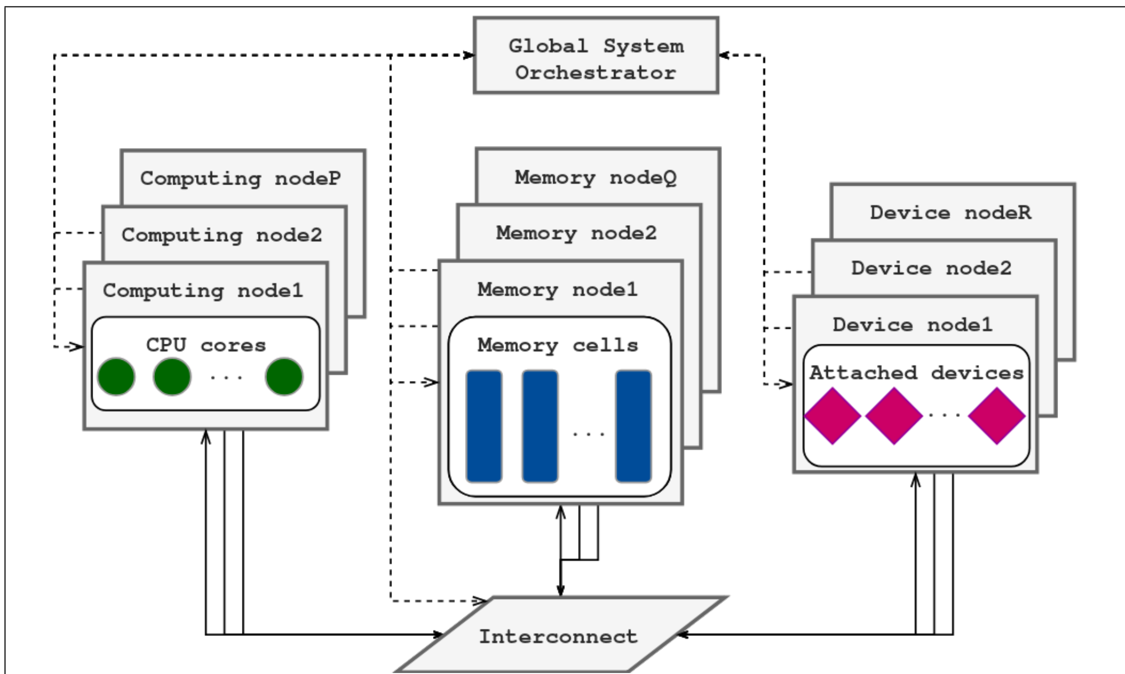


Figure 1-2: Disaggregated system architecture

Data-center resources may be divided into three main groups: CPUs, memory and peripherals [1, 35]. Instead of being contributed by a number of conventional monolithic

machines (as illustrated on figure 1-1), in a disaggregated architecture they are provided independently and form pools of resources. Each pool may consist of one or multiple nodes of a given type and all system nodes are attached to a reconfigurable *interconnect*, as shown on figure 1-2. It is therefore possible to update only one type of resources, for example by attaching more memory nodes. This is not a one-time improvement (like, for example, plugging more DIMM modules into vacant slots of a motherboard): such a flexible resource proportionality holds at scale. Figure 1-1 is a general illustration of the concept, in a particular implementation compute nodes may also be equipped with a limited amount of local RAM. This may be necessary to bring up a host operating system containing software modules responsible for registering the node to the system. What was being considered a machine in the context of conventional architectures essentially becomes a CPU node with the amount of other resources adjusted in a dynamic manner. As a result, the aforementioned notion of a slot no longer describes a monolithic unit characterized by a fixed resources proportionality. Instead, the slot is dynamically assembled now, with the amount of resources following requirements of a given workload. This requirements can vary over time, thus the slot capabilities can be also adjusted at runtime, for example by adding or removing devices or memory.

Thus, because of disaggregation, different types of resources can be reserved independently. For example, the problem of *memory capacity wall* (described in section 1.4) is bypassed as the amount of RAM required for a workload does not impose the amount of CPUs anymore because they are not physically bound to the same motherboard. Therefore, there are no CPUs which have to be reserved (and powered-on) only because of physical dependencies. In a disaggregated system the amount of reserved CPUs is only determined by the requirements of a workload. In this way disaggregation aims to improve the energy efficiency of the system by reducing energy wastes.

Such flexibility in resources allocation is a crucial novelty of the design that opens for more efficient balancing, less defensive estimation and finally increased utilization level, that means reduced energy losses. At the same time, reduced hardware dependencies make potential hardware upgrades (or failure recovery) easier and cheaper because a replacement of one type of resource does not enforce any changes to other resources.

Resource scheduler role

The relaxation of dependencies between resources is a crucial trait of the novel architecture for the aforementioned reasons. Nevertheless, it also makes the logic of the system scheduler more complex and the following section explains why.

Referring to figure 1-2, the scheduler is a submodule of the *Global System Orchestrator* (GSO). Being at heart of the system, the GSO is the only system component connected to all nodes via the *control network*, in order to pair selected nodes by establishing *interconnect* links. Additionally, it performs system-wide power management and provides a system administration interface.

With conventional architectures the scheduler has only to select a machine (of a fixed resource proportionality) and potential losses of underutilized resources are inescapable. This is changed due to disaggregation: instead of selecting a physical slot, the scheduler needs to dynamically define it by composing different resources together. At first, the scheduler has to select a computing node (an equivalent step to machine selection), on which a workload will be executed, and subsequently determine which memory and peripheral nodes should be attached to it, in order to meet the workload requirements. After computing node selection it may happen that other required peripherals are already attached. Otherwise, a proper new connectivity needs to be established between selected nodes. As a result, system resources may be used more efficiently but the scheduler logic is more complex. This is because the number of possible configurations is much higher (compared to a cluster of machines) and increases together with the number of nodes. However, in practice, there might exist boundaries limiting the number of combinations, related, for example, to a maximum allowed distance between two nodes or a maximum number of adjacent nodes that a CPU node can attach to.

Furthermore, because of the variability of the data-center load, new workloads can be deployed over time and others can terminate or get killed. Such fluctuations change values of parameters considered by the scheduler logic. In addition to proper resources arrangement at a workload deployment time, the scheduler has to monitor workload termination events in order to be able to dynamically reconfigure the system and keep the resource utilization factor at an acceptable level. For example, it may happen that after several applications terminated, there is only a few left running on a given CPU

node and it would be optimal to migrate them to another node and to power-off the now unused one. However, As this could temporarily slow down a workload execution, such a reconfiguration can probably not be performed too often: the utilization level benefits could be overshadowed by the operation overhead.

1.6 Disaggregated systems and virtualization

So far, this chapter discussed how the clustered system architecture looks like and what is the role of a virtualization layer in such systems. Furthermore, it was described what are disadvantages of the clustered approach and how the system architecture was disaggregated in order to eliminate them. Then, the question addressed in this section is whether the virtualization layer is still needed in a disaggregated system or not. Indeed, it is, and there are several good reasons to keep the additional level of abstraction between deployed workloads and the physical hardware.

The main assumption is that such a radical architectural shift as disaggregation is, while benefiting a system provider in many ways, should affect the way how the system is used as little as possible. From the user standpoint, the most important (in authors' subjective opinion) aspects of a cloud-computing service are: secure and stable workload execution (related to proper isolation between different workloads) and the lowest costs possible. As discussed in section 1.1, the costs are derivative of the utilization factor, which is further related to resource resizing flexibility. The improved resource arrangement flexibility, brought by disaggregation, encourages users to avoid overprovisioning and leverage the *pay-as-you-go* resource reservation model. The point is that users do not care about the system architecture; whether it is clustered or disaggregated, ideally they should be able to deploy their workloads the same way in both cases, in order to avoid additional engineering effort.

Therefore, similarly to clustered systems, virtualization is also an important layer of disaggregated ones. It abstracts underlying hardware and software infrastructure and provides a workload deployment unit that users are familiar with, that is a VM. For users, the crucial difference between clustered and disaggregated systems is that parameters of a VM (like memory volume or attached devices) can be adjusted more

flexibly or service price can become more competitive (no overprovisioning means less energy wasted).
Additionally, the level of a workload execution security should be at least the same as above. As mentioned in section 1.3, different workloads deployed as separate VMs are well isolated. A crashing workload within one VM is not be able to disturb others. Moreover, each VM runs its own software stack, for example with a customized operating system kernel. Such software dependencies are definitely an important decision factor for users considering migration of their business environments to the cloud.
Finally, as described in section 1.5, the job of a resource scheduler is more complex, in comparison with clustered systems. Nevertheless, thanks to the virtualization layer, the scheduler is able to perform resource management on a per workload basis. For example, a VM parameters definition allows to determine the optimal workload placement. Moreover, until a certain extension, a VM itself is able to perform a runtime monitoring of a workload execution. It can effectively support resources management across the system in a way, which does not require any modifications at the workload side. Similarly, the whole workload can be migrated to other system node simply by moving its enclosing VM. Otherwise, without the abstraction that virtualization brings, a system-specific support for migration would need to integrated in each workload, which would be an additional engineering effort at a user side.

1.7 Focus and scope of this work

This chapter introduced the context of this work, that is data-center systems. It presented the status of the sector and observed trends, with respect to workload characteristics. There were discussed inefficiencies of nowadays systems (like overprovisioning and low energy efficiency) and why they are related to clustered hardware architecture. In turn, there was presented a new architectural approach, which was postulated as a solution for aforementioned problems. Finally, this chapter justified why the virtualization layer is a crucial part of the clustered systems and why it will remain its important role also on disaggregated ones.
The point is that the virtualization layer needs to be modified in order to make the best

use of disaggregation benefits. Therefore, the main focus of this work is the adaptation of the virtualization layer to a disaggregated system, divided into several parts. They all depend on the assumption that there is a way to attach disaggregated memory to the host OS as well as there exists an interface to communicate with GSO in order to request for operations that affect other nodes of the system (like requesting for a chunk of disaggregated RAM, attaching a device or obtaining a shared memory lock). This work is associated to the project *dReDBox* and its scope presents only a fraction of all efforts conducted in the project[35].

The contributions described in the next chapters are related to flexible VM memory provisioning, inter-VM memory sharing, migration and disaggregated devices attachment. In order to position this work amongst related works from recent literature, respective analysis was performed and is presented in the next chapter. Although building the hardware and software components related to disaggregated memory provisioning at the host OS level is out of scope of this work, it is an important part which virtualization enhancements are built on top of. Therefore, this aspect is also present in the *state-of-the-art* analysis.

With respect to the term of *disaggregation*, the memory provisioning design presented in this work assumes that a VM is executed on a single computing node only (refer to figure 1-2). Although a VM can be migrated, this work is not dealing with VMs distributed over multiple computing nodes. It should be noted when comparing with other approaches, which assume the latter to be true[53].

Chapter 2

Related work

This chapter presents various works related to the subjects of disaggregated systems and virtualization. The presented positions are outcomes of the state-of-the-art analysis carried out by the author by putting in his best effort. The whole chapter is divided into five sections, according different topics, as indicated by the title of each section.

2.1 Memory disaggregation

This section presents in a high-level how the disaggregated memory is provided in the *dReDBox* prototype and how it compares to other solutions addressing the same problem. It presents methods allowing a CPU of one node to access the memory attached to another node. This ability is required to implement a disaggregated system but it can also be used in order to make use of underutilized memory in clustered one.

In general, the survey of the recent literature indicates that remote memory attachment may be done by software or hardware means. Orthogonally, a system may have either a traditional clustered architecture, or a disaggregated one, where different types of resources are provided by independent nodes. With respect to these two factors, all works discussed in the following part of this section were collated on figure 2-1.

	Page swapping	Direct access
Disaggregated	• Velegrakis2 • Lim1 • Shan	• Velegrakis1 • Lim2 • **dReDBox**
Clustered	• Hou2 • Samih • Svard • Nitu	• Montaner • Hou1

Figure 2-1: Categorization of works related to memory disaggregation

The labels correspond to names of first authors of discussed publications. In few cases the names are also followed by a digit indicating a variant of the presented approach (if there are multiple ones), in the order aligned with the description.

The presented approaches are assessed especially with regard to the level of system heterogeneity between computational and memory resources, the required hardware or software modifications as well as the support for virtualization they offer.

Velegrakis: *"Operating System Mechanisms for Remote Resource Utilization in ARM Microservers"*

Velegrakis [58] presented a small prototype, built with two nodes[1] connected by an FMC cable and a custom FPGA block called *Chip2Chip*. The custom hardware expands an on-chip interconnection in order to forward memory access transactions from the CPU of a node to the memory of the other (the disaggregated memory). The access can be performed in two ways.

The first one consists in reaching the remote memory through the CPU cache and it can be done using the cache of a local or remote CPU. The principle is similar to what is normally done for a local RAM, with one caveat that the cache can be remote. This access method is completely transparent for the host operating system, which only needs to know the physical address ranges associated to the disaggregated

[1] Avnet Zedboards, based on Xilinx Zynq cores that embed an ARM CPU and an FPGA fabric in the same chip.

memory. Once the host OS is notified about address ranges, the remote memory can be initialized as any other memory, although the author suggests that remote memory should be only used when the locally available one is full because of the performance overhead (latency, bandwidth). Zhang et al. took the same approach of attaching the remote memory at the on-chip interconnection level and accessing it at the cache-line granularity [64].

In our work the disaggregated memory is also attached transparently to the host OS and represented by physical address ranges but we took an alternative approach at the host OS level: the disaggregated memory ranges are initialized in a distinctive way. They are kept separate from the local RAM and not used by the default memory allocator in order to have a better control over the allocation of the disaggregated memory sections.

The same author also presented a second method of accessing the remote memory, which is using it as a swap partition. With this approach, the disaggregated resources are automatically used by an operating system to evict pages in case of high memory pressure. The most important difference is that disaggregated memory pages are not directly addressed but they are transferred back into local RAM by an I/O operation managed by the kernel swapping thread. Other works also use this technique. They are discussed in the following.

An interesting feature of this prototype is that the remote memory comes from a different node than the CPU node and the memory access can be handled completely in hardware (the first method). However, the prototype does not allow to test how the routing of the memory transactions would perform at scale, when a system would consist of multiple disaggregated memory nodes.

Although no specific virtualization support is mentioned, the system could be well integrated with the virtualization layer. It would be the easiest with the first access method, where the remote memory access is performed by the hardware, however the second one (swap device) is reported to perform better. In order to partition such a swap device amongst all deployed VMs, each one would need to obtain some amount of local RAM at first, where the data would be rotatively swapped-in. Although the author suggests that this is the way how disaggregated memory should be used (as a secondary resource), the work presented in this dissertation has been demonstrated on

a prototype (described in the following) based on different assumptions. To wit, in our work the deployed VMs are supposed to use only disaggregated memory, so the swapping technique cannot be used for the lack of local RAM.

Lim et al.: *"Disaggregated memory for expansion and sharing in blade servers"*

In [39], Lim et al. present a prototype of a system consisting of a disaggregated memory blade (equipped with a large volume of memory cells) connected to several compute nodes by a fast communication fabric. A distinctive feature of this design is that the virtualization layer is a complementary part of it. Each compute node runs a hypervisor that cooperates with the blade's management software in order to get assigned with segments of disaggregated memory. Subsequently, a hypervisor maps ranges of disaggregated RAM into its own physical address space, next to the local memory. Similarly to the previous work, two methods of remote memory provisioning are presented.
The first one is a hypervisor-level page fetching that copies remote data into local memory. It assumes that remote memory pages are marked as "poisoned" in the guest page table, in order to enforce a trap to the hypervisor on each access, trigger the remote data transfer and map the VM page to the fetched one. This method is reported to perform relatively well on average when a page fetching delay is distributed over multiple sequential accesses within its boundaries. If the memory access pattern is more random, the average performance decreases due to following reasons:

- More frequent page fetching require more exits to hypervisor, and more I/O operations that are expensive.
- Newly fetched data overwrite previous data, pointed to by some guest address. Therefore, a translation between the guest and hypervisor addresses has to be marked as "poisoned" again, because the same hypervisor page will now be reused to back a different address of the guest. This step requires an address translation table update as well as discarding the invalidated mapping from a translation cache, if it was there.

This approach would benefit from increasing the size of a local memory buffer assigned

to each VM but, with a given amount of RAM available in the hypervisor, the larger the buffer, the less VMs can run in parallel without overwriting each other's data. Moreover, in order to make use of disaggregated resources, this approach still needs locally available memory buffers, into which data can be fetched. A strong advantage of this approach, from an adoption effort point of view, is that it can be deployed with standard compute nodes; the only new component is the memory blade.

The second remote memory provisioning method presented is the remote access at cache-block granularity, which requires a custom hardware component in each compute node, redirecting *cache fill* requests to the memory blade. In return, no hypervisor modifications are necessary in order to access the remote memory: data fetching is performed by the hardware. Nevertheless, each access contributes a fixed delay to reach the memory blade. This is similar to local RAM accesses and therefore performance would greatly improve if the custom hardware component supported caching.

This work is very interesting because it presents a fully disaggregated system prototype with an integrated virtualization support. The system is composed of heterogeneous compute nodes and a common memory blade. Since the latter is a central element of the architecture, it can affect the system scalability in the most profound way, for example a maximum limit of addressable memory determines how many VMs can be feasibly deployed.

Montaner et al.: *"A practical way to extend shared memory support beyond a motherboard at low cost"*

A prototype by Montaner et al. [43] is also capable of expanding memory beyond the local motherboard by reserving subranges from other system nodes. A distinctive feature of this method is that it is based on the AMD HyperTransport technology [15], which supports a distributed (within a single motherboard) memory architecture out of the box. Each CPU has part of the physical memory available through a local controller and the rest is attached to other CPUs, all linked by a common interconnect. This work extends the interconnect in order to reach out to other system nodes. In addition to local memory controller(s), each node is also equipped with an additional controller used to redirect non-local memory accesses to remote nodes. Moreover, all of them

are assumed to be connected for the purpose of a memory reservation phase. While the reservation is handled by software, further accesses are performed completely in hardware. The high-level concept is similar to one presented by Velegrakis [58], Zhang et al. [64] and the *dReDBox* prototype [56], that is to perform memory transactions redirection at the on-chip interconnect level, with the help of a custom hardware. But differently to them, this work presents a system that is not disaggregated, in the sense that changing the amount of available memory entails changing the number of CPUs, as they are provided together by each system node.
Except for that, this work presents a system that helps to improve global memory utilization level by sharing non-used memory regions between different nodes. Although it is not discussed in the paper, this system design does not prevent virtualization layer integration. Only the memory reservation phase is performed by software and it could be performed at VM boot time. Then, further accesses to remote memory are completely transparently from the software perspective. Perhaps the only indication of reaching the remote memory may be a lower access performance.

Hou et al.: *"Cost effective data center servers"*

Hou et al. proposed a system leveraging the PCIe SR-IOV[2] in order to share memory across nodes[30]. The prototype consists of one *root* node connected to a PCIe switch through a *transparent bridge* (TB) port and four other *leaf* nodes attached through *non-transparent bridges* (NTB). The root node serves as a *root complex*, in terms of the PCIe standard. Each node runs an OS with a custom driver responsible for initializing address translation mappings in NTB ports and exposing a simple interface to applications, providing `remote_alloc` and `remote_free` calls. A given memory region can be accessed by exactly one node at any given time and the access can be performed in two ways.
The first one consists in direct relaying of each load/store instruction. At first it happens at the local memory controller and further the instruction is dispatched by the PCIe switch. The performance of this method is reported as not good due to the configurations steps performed by software as well as the lack of data caching.

[2]Single-Root I/O virtualization, an extension of the PCI Express standard

Alternatively a batch transfer may be configured, using the DMA engine located at the NTB port. It copies the data and emits an interrupt to the sender on completion. In this mode, the remote memory is exposed as a virtual block device, which can be used as a swap partition in order to extend the local RAM. This configuration offers much better performance, in comparison with the first method. This is another hybrid approach, combining both local and remote memory to be used by a host process (for example a VM).
A primary objective of this work was to design a system offering effective resource sharing built with commodity equipment, no custom hardware components were used. In terms of scalability, the *root* node may impose a maximum number of attached system nodes. Although the virtualization aspect is not discussed in this work, similarly to the work presented by Montaner et al., a swap space could be partitioned between all VMs deployed on a given host and the remote memory reservation phase integrated with the virtualization layer. Also, this approach allows to share memory resources between different system nodes but they are not disaggregated, the proportionality between computing and memory resources is fixed as they are provided together by each system node.

Samih et al.: *"A collaborative memory system for high-performance and cost-effective clustered architectures"*

Samih et al. presented a work, in which nodes of a cluster collaborate together in order to share memory across each other [51]. There is a software management layer distributed across the system, which leverages a dynamic negotiation protocol. It allows a node to request for memory from other nodes as well as offer its own memory for them. A node can also perform none of this and stay in a neutral state. As mentioned, it is a dynamic protocol and therefore the state of a node may evolve at runtime, according to the current memory pressure.
When reserved, the remote memory is exposed to the host OS as a swap partition and the underlying data transfers between nodes are performed as DMA operation. In this aspect, this the same approach as the batch transfers approach of the work by Hou et al.[30]. The difference is that the interconnect type is Ethernet and the communication

protocol is TCP/IP, however the system could be as well based on InfiniBand or PCIe. With regard to the virtualization and disaggregation criteria, the same comments apply as for the previous work [30]. Moreover, with regard to scalability, it should be noted that the convergence time of the negotiation protocol may increase with the number of system nodes.

Svärd et al.: *"Hecatonchire: Towards Multi-host Virtual Machines by Server Disaggregation"*

Similarly to previous works ([30] and [51]), the system proposed by Svärd et al. also supports remote memory sharing between nodes based on DMA transfers. However, differently from them, it is well integrated with the virtualization layer [55]. The memory from other nodes is provided to the host (hypervisor) as a block device but it is then abstracted in the VM by a guest address space. Upon guest's access, if the accessed page is not already present locally, a page fault is generated. Subsequently, a custom host module handles it by fetching the page from the remote location and mapping it properly to the guest. Thus, this approach is very similar to the one presented by Lim et al. [39] and holds the same characteristics.

Nitu et al.: *"Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter"*

In the paper by Nitu et al. there is presented another approach dedicated for clustered systems, which allows to access memory of another machine by leveraging RDMA technique, here over Infiniband. Each machine can consume memory resources of other machines or donate its own RAM. It can also make an exclusive use of its own RAM and not expose it to others. Such roles are assigned by a global management layer and can be changed dynamically. Pages of remote memory are provided as a swap in two variants, either to hypervisor or directly to a VM. In the second case a page is also swapped to a VM-local disk as a fault tolerance mechanism. In a high-level of description, so far this work seems to be similar to the previous one by by Svärd et al. [55]. Indeed, in some aspects it is. However, a distinctive trait of the paper is the focus

on energy efficiency.
Authors stress that the shift to a disaggregated architecture is a fundamental change at the bottom of the system. Therefore, they proposed a solution that can be successfully adapted to traditional clustered systems as a half-way solution. Instead of enforcing a physical hardware disaggregation they suggest to disaggregated memory and CPUs logically, at the power supply level. It means that the power management logic can handle both types of resources independently and reduce energy wastes in this way. They developed a novel *"zombie"* state of a power management framework, in which the whole board is powered-off except for its memory banks and part of network. Thanks to that, the memory can be accessed by other system nodes while consuming much less energy, comparing to the situation when all board components are active. This paper provides a nice way of improving energy efficiency in already deployed systems. Moreover, it points out an important matter, that is applicability. Large-scale systems are expensive installation, thus changing radically the architecture may not always be affordable. In such situations evolutionary modifications seem to be preferable, not revolutionary ones.

Shan et al.: *"LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation"*

Shan et al. predict that future data centers will adopt the concept of hardware resource disaggregation and propose a dedicated operating system for them called *LegoOS* [53]. In a high-level description, it is composed of multiple single-purpose nodes installed, meaning that each node provides only one type of resources. There are processing, memory and storage nodes, interconnected with a network in order to build up a fully functional system. Similarly to other mentioned positions, there is a global management layer orchestrating the whole architecture.
The operating system, proposed by the authors, has a *splitkernel* structure, which means that it is distributed over multiple heterogeneous modules called monitors, with names associated to managed hardware nodes. A monitor acts locally to perform its purpose-specific tasks as well as it exchanges messages with others in order to use remote resources. From a workload perspective resources are visible as a virtual server. Each

virtual server can be composed of multiple different monitors, as well as each monitor can be used by more than one virtual server.
Comparing to all other works presented, a distinctive feature of this design is that one process, for example a VM cannot know on which processing node it will run. Different processes of the same application may be executed by different computing nodes. Such transparency stands out this paper as the one of the highest degree of disaggregation. Other literature positions described in this chapter present how to leverage remote memory or devices but with execution of a workload always bound to a single processing node (except for explicitly distributed application which in this sense can be seen as separate workloads). Therefore, the work by Shan et al. being a reference, it should be stated that all other publications mentioned in this chapter do not present the same level of disaggregation. Nevertheless, usually such information is not explicitly specified and can be only derived from the description, thus the categorization may be slightly ambiguous. All in all, instead of abstracting the hardware by VMs bound to a single processing nodes, the *LegoOS* proposes concept of virtual nodes split over multiple hardware components. Provided the network performance will significantly improve in future, such approach is a very interesting alternative to traditional virtualization.

Syrivelis et al.: *"A Software-defined Architecture and Prototype for Disaggregated Memory Rack Scale Systems"*

The *dReDBox* project [56], which this dissertation is associated with, applies concept of memory disaggregation similar to the one found in the works by Velegrakis [58] and Zhang [64].
Reaching the remote memory is similarly done by forwarding on-chip interconnection transactions to remote nodes. This is transparent to the software stack; from its perspective the disaggregated resources are directly accessible as ranges of physical address space. The *dReDBox* paper [56] presents the first prototype version, equipped only with one compute node and one memory node. However, at the time of writing, the system is ready to receive multiple nodes of each type. This makes it much different from the other referred works, in the sense that it presents a fully-disaggregated and scalable system with resources provided independently to each other. Moreover, in

addition to the startup configuration, sections of disaggregated memory can also be added or removed at runtime in a *hot-plug* manner. The connections between the nodes are established dynamically by the system orchestration logic.
A practical limitation of both aforementioned works is that they are based on 32-bit platforms (also by ARM), which are not very interesting in a data-center context because of the limited amount of directly addressable RAM, far less than required nowadays by server workloads. Reportedly, in this sector, at least 95% of market share is currently owned by Intel (as of 2018) and other chip manufacturers hope to increase the popularity of 64-bit ARMv8 platforms [26]. For this reason, it is more relevant to see the disaggregation concept implemented on this architecture. Moreover, porting the software stack from a 32-bit to a 64-bit ARM platform is not a straight-forward task. Other domains, where 32-bit platforms are more widespread (IoT devices, mobile devices, automotive), do not currently seem to require the large amounts of memory that would justify disaggregation.

Summary

Regarding remote memory attachment, the first alternative refers to the question whether a remote memory cell is attached to the same motherboard (or integrated in the same chip in case of SoC platforms) as the CPU or not. In the second case, the local on-chip interconnect needs to be bridged with the remote one, as in case of *A1, B2, C, D1* and *I* on the figure 2-1. This seem to be the most elegant solution as it directly addresses the root limitation and is completely transparent from the software point of view at the same time. Nevertheless, in practice these approaches may be characterized by inferior performance, in comparison with swapping approaches, as shown in *A, B* and *D*. That is because a direct access is performed usually at cache-block granularity by issuing a local interconnect transaction. Each transaction introduces a delay related to the distance between two nodes as well as interconnect packets processing and routing. This delay is accumulating for multiple subsequent direct accesses. This situation could perhaps be improved with proper cache hierarchy scaling and prefetching, as it is for local memory accesses, where the difference in access latency between a first level cache (L1) and a memory cell is typically larger than an order of magnitude (for

example, for a typical PC it is about 5 ns vs. about 100 ns).

If not by hardware, data from remote memory is fetched in a page granularity by software means, as shown in *A2, B1, D2, E, F, G and H* on the figure 2-1. This may offer better performance, especially in case of sequential memory access patterns, as the penalty of copying one page of disaggregated memory will be distributed over multiple cache blocks it contains. Another characteristic of this approach is that it does not require any hardware customizations, which can be considered as an advantage. For example, modifying hardware that is already deployed may be difficult, expensive and requires a lot of engineering effort, in comparison with a software-based solution. On the other hand, as the name *swapping* indicates, this method requires a certain amount of local memory pages to serve as a destination buffer for data incoming from the remote memory. The bigger the disproportion between the size of local and remote memory is, the more often local pages will be overwritten. If there are too many overwrites at the same time, it may happen that all system threads will be waiting on I/O operations to complete and the guest will be idle at that time. This situation is known as *memory congestion* and it degrades the system performance.

In general, remote memory access techniques are mandatory for scalable disaggregated systems composed of heterogeneous nodes of different types. Traditional clustered installations may also take advantage of them in order to make use of underutilized memory of other machines and improve the global memory utilization factor. Moreover, as presented by the position *G* of figure 2-1, it is also possible to improve the *energy efficiency* of existing clustered systems by smarter power management, thus without the need discarding existing hardware in favour of disaggregated systems. Indeed, from the practical point of view the latter is a very important remark.

Works *B* and *F* of figure 2-1 show that either way of reaching the remote memory can be integrated with the virtualization layer. Though, with the swapping approach, each VM also requires a certain amount of local memory for the reasons described before. Assuming that each host would run multiple VMs, this may actually create a requirement of a significant amount of RAM being available locally, while the remote memory will be considered a secondary resource. Instead of implementing such a hybrid system, the *dReDBox* project (*I* on figure 2-1) aims for a higher degree of

heterogeneity, in the sense that locally available RAM is only supposed to be used for the host OS purpose but all VMs will be using the remote memory obtained from disaggregated nodes. For this reason the project prototype employs a hardware attachment method so that the disaggregated memory is visible from the host OS perspective as ranges of a physical address space. The ranges are also isolated from the standard host memory allocator, which will be described in Chapter 3 presenting the virtualization layer memory provisioning.

2.2 VM memory provisioning and balancing

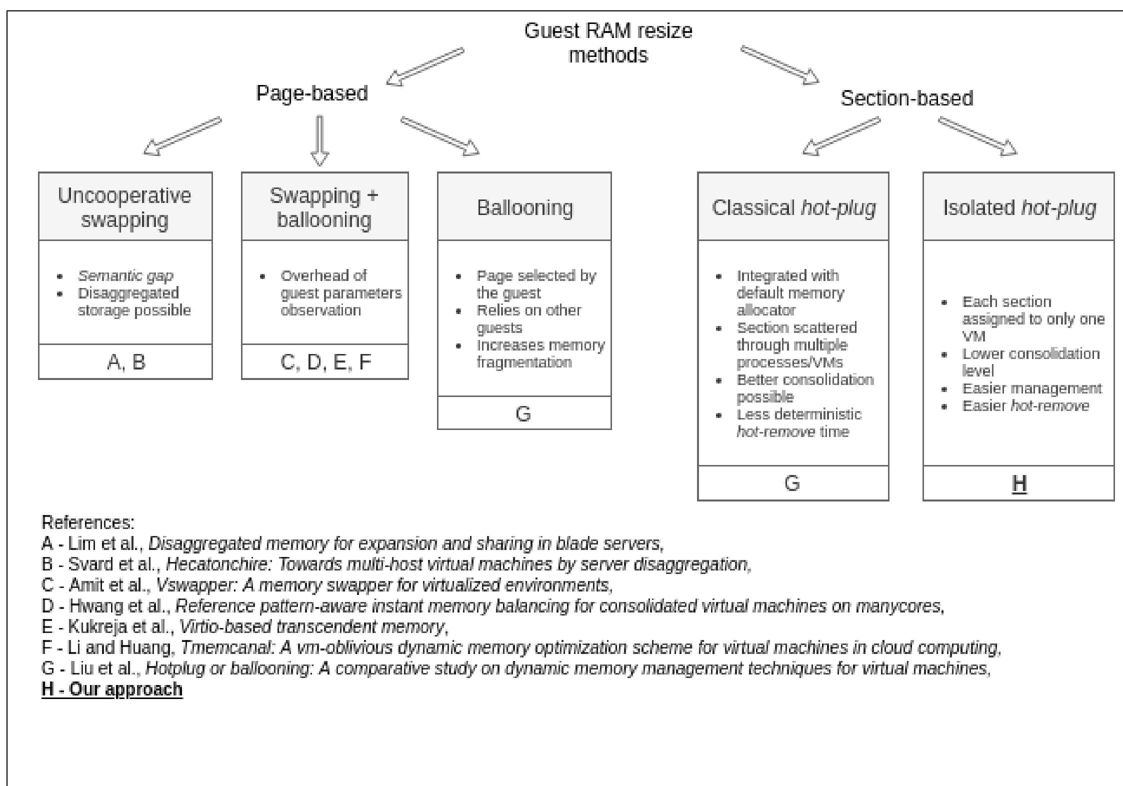


Figure 2-2: Comparison of runtime guest memory resizing methods

In this section, there are discussed different ways of guest memory resizing at runtime. For an illustrative purpose, they are all collated on the figure 2-2, together with the most important characteristics as well as example positions from recent literature, which leverage a given approach.

A comment on assumptions of our project

Based on the *dReDBox* prototype, capable of allocating memory coming from the disaggregated pool, one of our main objective was to enhance the virtualization framework so that hosted VMs would take full advantage of the non-local resources.
The idea is that a compute node may have attached locally only a very limited amount of RAM, necessary to boot the host OS (serving as a hypervisor) and to execute auxiliary software components. As mentioned at the end of Section 1.7, such approach may be considered as *semi-disaggregated* for two reasons that follow. Firstly, the hypervisor itself relies on a certain amount of memory being available locally on the computing node. Therefore, such requirement bends a little the principle of a complete independence between CPUs and RAM in a disaggregated system. Secondly, although the attached memory comes from remote nodes, a VM process is bound to the HPA of a single computing node as is the hypervisor. For such reason, a VM can be migrated to a different computing node but it cannot be distributed over multiple ones at the same time. Different nodes run different hypervisor instances. This is way different comparing to the *LegoOS* design, where the hypervisor is built on top of a *splitkernel* architecture [53]. Thus, although our approach as well as similar ones are often described in the literature as *disaggregated*, the level of disaggregation is different when compared to a system like the *LegoOS*.
Additional processes executed by the hypervisor next to VMs are required to integrate the compute node with the rest of the system. For example, the host OS is notified by the system manager when a VM process should be started or migrated. Moreover, a compute node also needs a way to request the attachment of a portion of disaggregated memory and receive the corresponding parameters in return, in order to properly configure its local interconnect adapter logic and establish a link to the new resources. These tasks are handled by software components running in local RAM for the obvious reason that the disaggregated memory cannot be used before it is attached. Differently to that, all the buffers supposed to build up the guest's RAM come only from the disaggregated pool.
Building on top of these assumptions, the virtualization framework is supposed to obtain buffers of disaggregated RAM and attach them to a VM at its boot time. Corre-

spondingly, they are released when a VM is terminated. Moreover, VMs are expected to dynamically change the amount of provided RAM at runtime so that a running guest may increase or decrease its memory capacity without reboot. This functionality underlies the memory balancing logic at the host level. The host can shuffle already reserved buffers between all hosted VMs and, when these are not sufficient, it can request the attachment of additional ones.
In this section, the problem of memory provisioning and balancing is just introduced in order to position it properly between similar *state-of-the-art* approaches. Chapter 3 is devoted to the detailed presentation of how the guest's RAM is provided and how its dynamic resize is implemented. Other works are not necessarily deployed on disaggregated systems because memory balancing between VMs operates on ranges of HVA. Whether these ranges are linked to local memory banks or remote ones is not relevant for the balancing mechanism.

Overcommitment

Most prior works related to guest memory balancing implement different flavors of page-based techniques, also referred to as the *overcommitment* approach. Its principle of operation is that, at any given moment, only a fraction of a guest's RAM is actually mapped to physical resources owned by the host. While the guest is running, these mappings are dynamically modified in order to support the *working set* of the guest, that is the set of pages it is actually using.
The most straight-forward method is the *uncooperative swapping*, that enables overcommitted pages (beneficiary) to be accessed at the cost of moving victim pages to swap storage in order to release physical resources. Therefore, during guest's operation, memory pages are constantly juggled to back up currently used *working set*. Since it is performed by the hypervisor, this technique suffers from a so called *semantic gap*, meaning that the hypervisor cannot optimally select guest pages to be evicted for the lack of precise information bout the guest memory usage [20, 21]. Another drawback, from the performance perspective, is that each page replacement requires a context switch to the hypervisor in order to perform an I/O operation.
It should be noted that examples employing this method were also mentioned in Sec-

tion 2.1. The first variant presented by Lim et al. [39] as well as the work by Svärd [55], are both operating according this principle, however they are using the disaggregated memory, instead of a disk, as a swap storage.

Ballooning

Ballooning was proposed to address the *semantic gap* problem by delegating the page selection process to the guest [59], which reserves a page of memory using a standard memory allocator and passes its address to the hypervisor. The latter translates the address into a valid one in the host address space and marks a page as temporarily released by a given VM (it can later be claimed back). Using a standard memory allocator implies that the guest *balloon* driver has no direct control over the physical address of the selected page. Although the guest has access to all statistics to make an optimal choice from its performance perspective, this method has other drawbacks. A VM that needs more pages relies on other VMs on the host to relinquish them but it is not guaranteed to succeed as the *ballooning* scope can be limited or disabled by other guests. Moreover, other guests may additionally need to store data from some pages to disk, before they will be able to release them. Therefore, *ballooning* is less deterministic and can be slower than *uncooperative swapping*. It cannot help rapid bursts in memory needs [32], but, on the other hand, the *uncooperative swapping* may also degrade the performance if executed frequently.

Combined approaches

Several combinations of both mechanisms have been proposed to select the best candidates for eviction as well as to improve the balancing responsiveness [32, 21, 36, 38]. For example, it is preferable to evict guest pages keeping data that were read from disk (this is known as the *page cache*). They are already present in a persistent storage so the step of dumping page contents is spared.
Nevertheless, all page-based techniques operate within the range of guest memory defined at boot time. Thus, they are able to reclaim pages already owned by a VM but cannot obtain more memory than it was launched with. Moreover, these techniques are also known to introduce guest memory fragmentation. That is because they are

reserving pages without any control over their physical address (decided internally by the memory allocator). For this reason, after running for a long time a guest OS can become unable to find a range of contiguous memory to fit a large object [40, 52].

Expanding declared RAM capacity

Up to the author's best knowledge, only one paper by Liu et al. [40] combines the *memory hot-plug* mechanism (that is *hot-add* and *hot-remove* operations) together with *ballooning*.

The memory *hot-add* expands a system's physical address space, at runtime, by a contiguous range of addresses called *memory section*. A section of memory corresponds to attached memory resources in the sense of hardware connectivity (or virtualized hardware, in case of VMs). Symmetrically, memory *hot-remove* does the opposite operation. The *hot-add/-remove* operates on a memory section granularity, which length is a platform-dependent parameter (for example, in Linux it is 1GB by default for the ARMv8 architecture). This is a coarse-grained resizing mechanism, compared to page-oriented methods, as one section of memory may contain hundreds of thousands of pages (the page size is typically 4KB, although 64KB is more and more common for systems with large amount of RAM). The *hot-add/-remove* perform actual resource attachment/detachment in the same way as at VM boot time or termination. This is crucially different from swapping pages, as with *overcommitment*, or from marking them as unavailable, as with *ballooning*.

The authors of the aforementioned work extended the *Xen* virtualization framework with a component responsible for periodic collection of runtime memory usage statistics in all running VMs. Based on these observations a decision about a VM memory resize is made. Primarily, the resize is performed on a section granularity. Since only an integer number of sections may be added, if the attached resources are bigger than the requested amount, the outstanding pages are set as *offline*. In this state they are not usable for the guest memory allocator and can be reclaimed by the hypervisor in a *ballooning*-like way. The reclamation process starts from the highest page index (an upper part of a virtual address). Moreover, if not used by other guests, the same pages can later be quickly switched to *online* state in order to increase the guest's available

memory in a fine-grained granularity. This hybrid system allows to maximize the host's memory utilization level better than by memory *hot-plug* alone. At the same time it does not drastically increases the guest memory fragmentation level (as with classic ballooning).

Approach presented in this work

Similarly to the work by Liu et al. [40], the virtualization framework presented in Section 3 also takes advantage of a section-based memory resizing. Nevertheless, in our system the VM memory provisioning is tightly coupled with memory disaggregation. Sections of disaggregated memory are attached **to the host** as isolated, that is not passed to the default host memory allocator. Subsequently, the system presented in this dissertation does not take advantage of the host memory allocator. Instead, at VM boot time, the virtual RAM of a guest is constructed from one or multiple isolated chunks. Because of that, each guest memory section is also contiguous in a host physical address space, and each isolated chunk at host level is always attached only to one VM.
This guest memory provisioning design simplifies the integration with a disaggregated system. Neither do we employ any periodic guest memory usage statistics collection nor *ballooning*, although the system does not prevent that. The rationale behind this choice is that it would introduce additional complexity to the system, while, in the light of having a (supposedly very large) disaggregated memory pool, it is affordable to loose the fine-grained memory regulation in order to keep the host memory fragmentation low. Page-based balancing methods are only secondary techniques, they cannot overcome a maximum guest memory size declared at boot time, as mentioned above. Moreover, classic *ballooning* may block a guest memory section from being unplugged until the conflicting page owned by the *balloon* driver will be migrated to one from other sections. The process of looking for a replacement page imposes an additional indeterministic performance overhead. This problem could be tackled by limiting *ballooning* to only one isolated section of the host, shared by multiple VMs. This section would need to be always present in all guests. But on the other hand, it would require a dedicated allocator used by the *balloon* driver and synchronized with the host side. This allocator would only operate within the boundaries of the specific

section.

In the design presented later in Section 3 this problem is avoided. At each VM termination, it is always known which isolated chunks of disaggregated memory a VM was using and these chunks are released. It is also guaranteed that these sections were not used by any other VM (except for explicitly shared sections, which are one of the subjects of Chapter 4). On the other hand, if, for any reason, a host is supposed to release a particular chunk of isolated memory, there is always only one VM affected.

The total host memory utilization factor may be lower than it was with page-based balancing methods included, but, provided an expectedly large amount of RAM available in a disaggregated system, this is most likely acceptable. Additionally, this simplifies the management of the disaggregated memory, improves the determinism of the resizing delay and does not introduce an excessive memory fragmentation in the host.

The presented approach is not a magic bullet, it is not supposed to handle instant bursts in memory demand but *ballooning* also does not do it, while bringing the other difficulties mentioned above.

2.3 Uniform address space

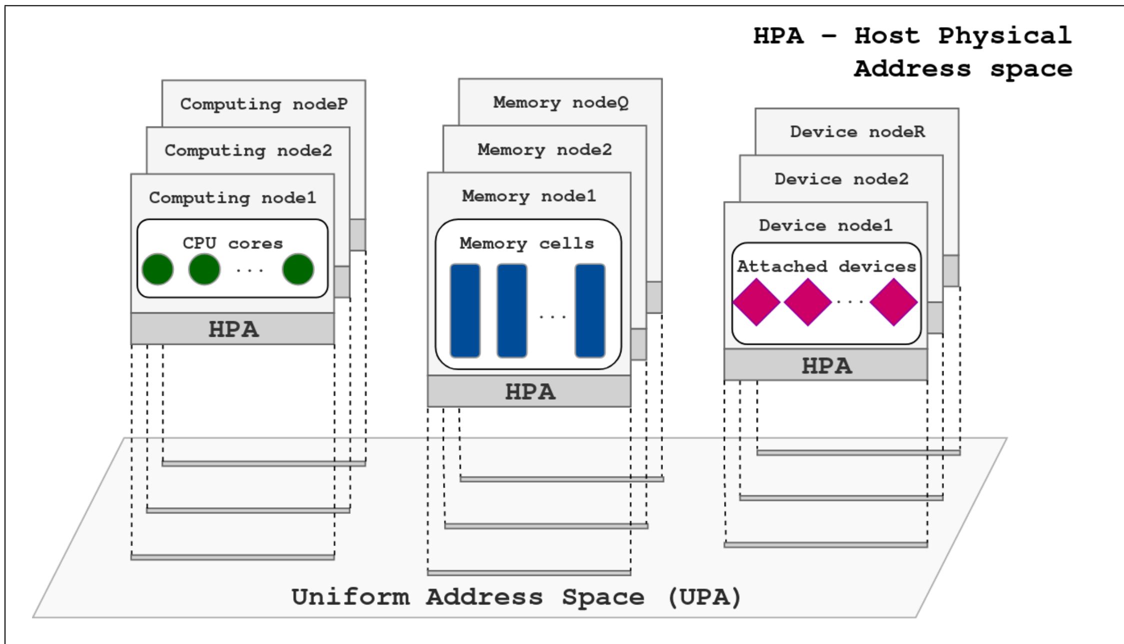


Figure 2-3: Simple illustration of a Uniform Address Space concept

Except for memory provisioning to virtual machines, in this dissertation we also discuss the concepts of memory sharing between VMs, VM migration and device pass-through assignment — all in the context of a disaggregated architecture. These ideas are presented using a global picture of the system, in which all available resources are described by the *Uniform Physical Address space* (UPA), simply illustrated on figure 2-3. This is only one of many possible ways of illustration, it does not mean to express any particular relations between UPA ranges mapped to particular resources.

Typically, the different resources of a given system node are described by distinctive ranges of a physical address space (referred to as HPA on the picture) and the layout is usually determined at hardware production stage. In a disaggregated architecture the multitude of separate nodes makes such a description ambiguous as the same HPA range may refer to different resources at different nodes. Therefore, in order to be able to uniquely index all resources available across the system, they are mapped on a (possibly very large) *uniform address space*. This mapping can be, for example, managed by the *global system orchestrator* when a node is first registered. The mapping should describe all resources of a node that are globally relevant, especially the ones that are going to be shared with other nodes (memory and devices). This is because a pair of UPA addresses allows to describe an interconnect link. This approach is not new, regardless of the actual term used to name it. It simplifies the description of various mechanisms affecting multiple nodes, while the complexity of an actual implementation is hidden [22, 43].

2.4 Inter-VM memory sharing and migration

In Chapter 4 we will discuss inter-VM memory sharing and VM migration on a disaggregated system. Both are presented from the virtualization framework perspective, building on top of custom libraries available at host level. The libraries are responsible for operating parts of system-wide infrastructure involved in memory sharing or migration.

Inter-VM memory sharing is mostly mentioned in the literature in the sense of communication and data exchange between VMs co-located on the same host. On a traditional

architecture this allows a *zero-copy* sharing as data is all the time in the same physical location, mapped to all participating VMs. In this context the main challenge is to assure proper serialization of concurrent accesses in order to avoid data corruption. The related signaling is usually held by a server running in the hypervisor and dispatching the virtual interrupts between the participating VMs.

First author	**VM abstraction**	**User interface**	**Comments**
Huang [31]	Virtual device	Socket	Standard communication interface optimized for co-located VMs
Mohebbi [42]	Custom PCI device	POSIX shmem	Sharing between VMs as if between guest processes
Kurtadikar [37]	IVSHMEM	mmap	Switching to TCP/IP for VMs on separate nodes, migration breaks performance
Ivanovic [33]	IVSHMEM	MPI	Optimizing parallel framework with IVSHMEM
Zhang [63]	IVSHMEM	MPI	Optimizing parallel framework with IVSHMEM
Nitu [45]	N/A	N/A	Memory sharing not discussed explicitly but not forbidden
dReDBox	Custom PCI device	POSIX-like	Unified memory sharing regardless of VM co-location

Table 2.1: Inter-VM memory sharing methods

The table 2.1 puts together a set of works proposing differnt methods of sharing memory between different VMs. They are also described in a slightly more detailed way in the following.

Except for *dReDBox*, all presented examples are using *zero-copy* memory sharing only when all participating VMs are running on the same system node, that is using the same HPA. In one case this approach is used to provide an efficient socket-based communication, not affected by the overhead of a networking stack [31]. Other works integrate it with HPC programming models (like MPI) in order to optimize inter-VM communication for co-located VMs [33, 63]. Nevertheless, because of clustered architecture limitations, once VMs are hosted by different system nodes they fall back to mechanisms based on data copying, that is TCP/IP or *Remote Direct Memory Access* (RDMA). The latter can be over PCIe, as in [30, 57], or InfiniBand — very popular in HPC domain [63]. Amongst them, the most efficient is when copying is initialized by software but the actual data transfer is performed only by hardware.

The same applies, for example, when one VM, initially capable of *zero-copy* data

sharing with others, is migrated to another node. The data sharing scheme has to be changed to a copy-based one and the data exchange performance will decrease as a result. In such scenario VM migration is considered harmful, while ideally it would be almost transparent to the deployed workloads.

Huang et al.: *"Virtual machine aware communication libraries for high performance computing"*

One approach, by Huang et al. [31], is based on the *Xen* virtualization framework. It establishes memory sharing through the *grant tables* mechanism. It allows a VM to share its own memory regions with other participating VMs. They can see the shared region as a virtual device. The device contains a data section divided into several fixed-size chunks and a ring-buffer with the same number of slots, used for access synchronization. The mechanism supports socket-like reads and writes and therefore it can be transparently used by applications leveraging standard network communication between VMs.

Mohebbi et al.: *"Zivm: A zero-copy inter-vm communication mechanism for cloud computing"*

In another work by Mohebbi et al. [42], shared memory is provided to guest applications through the standard POSIX interface. Practically, it eases the application development process as it allows to test an application on a single machine and then deploy it on a target system without changing the corresponding code. A shared host memory region is virtualized as a standard PCI device and, for better portability between different hypervisors, communication with the guest is done over a VirtIO channel.

IVSHMEM-derived approaches

Eventually, several authors leveraged the IVSHMEM technique, which exposes shared memory obtained from the host as a PCI device in the VM. Each such device contains the respective *Base Address Register* (BAR) registers representing the shared regions [12]. With the help of the IVSHMEM-server running on the host, the access

shared between several VMs hosted on the same system node can be properly synchronized [37, 33, 63].

Nitu et al.: *"Welcome to zombieland..."* - migration aspect

As described in Section 2.1, the solution proposed by Nitu et al. [45] postulates a logical disaggregation between memory and CPUs, performed at power management level in order to improve memory utilization and energy efficiency of clustered servers. By introducing the novel *zombie* power state this approach allows to use remote memory of other nodes, even if their CPUs are powered-off.
There is also discussed the VM migration aspect in this position. Although another power state complicates the migration task, this work mentions the benefits of having part of guest's RAM located at remote node. In such case, this part does not have to be moved anywhere, only the VM before and after the migration needs to point to the same remote resources. Moreover, the publication explains how it can be integrated with live-migration mechanism.
In comparison with the *dReDBox* approach presented in this dissertation, in the latter, there is a broader discussion of the distinction between local and remote part of VM memory footprint as well as expected benefits for the resource management. On the other hand, the subject of integration with *live-migration* techniques is only mentioned in proposed future works.

Summary

In Chapter 4 we will discuss how the architecture disaggregation affects both memory sharing between running VMs as well the notion of VM migration. Up to the author's best knowledge, no position in the recent literature covers these topics. Nevertheless, thanks to the fact that CPUs executing VMs and their disaggregated memory are located at separate system nodes, the inter-VM memory sharing can always be implemented in a unified *zero-copy* manner. A particular case of having associated VMs executed by the same node creates only additional optimization opportunities. For the same reason of disaggregation, moving a VM that uses a shared memory region from one computing node to another affects only the VM itself. In particular it does not affect other VMs

participating in a sharing group. Moreover, it does not change the way of accessing the shared region, especially no additional data copies are necessary. In this context, the migration (referred to as CPU migration in further chapters) is not harmful anymore for established sharing instances. Eventually, regarding the term "migration", we present how it gains a twofold meaning in the light of disaggregation, with respect to CPU and memory resources, which are migrated independently.

2.5 Devices disaggregation

A (Hou et al.)	B (Tu et al.)	C (PCIe extensions)	D (Suzuki et al.)
• Host-to-host DMA done by NTB port • Additional software emulation required • Based on commodity hardware only • No disaggregation	• Devices attached to a single tree of the MH • Translation tables of each NTB programmed at device enumeration • Each host gets assigned VF(s) • Disaggregation limited by the switch	• **SR-IOV**: single RC • **MR-IOV**: real disaggregation, limited HW availability	• Distributed SR-IOV switch • SW/HW abstraction - PCIe over Ethernet • Custom interconnect adapters

E (Shan et al.)	**F (dReDBox)**
• Full disaggregation • Managed by distributed OS • Generic type of device attachet to a *monitor* • Custom HW component at device node • Network-based communication	• Full disaggregation • Supports legacy and SR-IOV devices • Requires *GSO* cooperation • Custom interconnect adapters

References:
A - Hou et al., *Cost effective data center servers*,
B - Tu et al., *Marlin: a memory-based rack area network*,
C - PCI Express standard extensions,
http://www.cs.nthu.edu.tw/~ychung/slides/Virtualization/SR-MR-IOV.pptx,
D - Suzuki et al., *Multi-root share of single-root I/O virtualization (SR-IOV) compliant PCI Express devices*,
E - Shan et al., *LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation*
F - Our approach

Figure 2-4: Hardware-based remote device sharing methods

Analogously to the previous sections, this one presents an overview of relevant literature positions discussing the subject of device disaggregation. Related works are graphically collated on the figure 2-4 and further described below. Not all of them are implemented on disaggregated architectures but they could be adapted appropriately and therefore it is useful to have a look at them. In all cases, the common main purpose is to allow an application running on a computing node (also inside a VM) to use a device that is not directly attached to the node. A reason for that is to have less devices installed across the system but keep them busy all the time instead of attaching a given type of

peripheral to each node, even if it would be underutilized.

Hou et al.: *"Cost effective data center servers"*

The PCIe-based system by Hou et al. [30] also presents a method for devices sharing in two examples: a general-purpose graphic card (GPGPU) and a network interface card (NIC).
A GPGPU is a great example of an accelerator that only a limited set of workloads can effectively use, while it is marked by a non-negligible price tag. However, for applications that are indeed capable of making use of it, the performance can be significantly improved. Because of that, it is recommended to have at least few such devices installed and shared across the system.
In Hou et al. [30] work sharing is based on exposing address ranges, corresponding to memory-mapped registers, to other nodes through an NTB port. Additionally, input and output data buffers have to be allocated at both nodes and the data needs to be copied (by DMA engine of an NTB port) before and after operation of the device, respectively. Therefore, this method of remote device attachment is almost identical to the memory sharing scheme presented in the same work. The biggest drawback of this solution is that input data needs to be copied to a node that has the device directly attached, before its operation can be launched. The same happens to output data after processing is completed. This is because the device cannot directly operate on ranges coming from an address space of a remote node. Moreover, it is possible that some operations would not require the processing of the whole input. In such cases a fraction of the time spent on data copying would be wasted. Instead, it would be perhaps more efficient if such a device could fetch input data directly from the remote node on a per-need basis and could transfer the output along the operation. Additional optimizations would assume that copying unmodified bytes from input to output buffers could be also performed directly on a remote node's memory, without transferring it back and forth to the node that has the device locally attached.
The second presented example figures a shared *network interface card* (NIC). In order to access a remote NIC, each node has an associated virtual NIC (vNIC), emulated by a driver and having a unique MAC address. The software emulates the IP layer over

existing PCIe links. Each vNIC has its own routing table that associates the MAC address of a destination card with the proper PCIe link. Similarly to the previous example, data transfers between vNIC and NIC are performed by the DMA engine of an NTB port. The weakest point of this approach is that the emulation is done completely by software and that additional data copy between the vNIC node and a NIC node is required. A device cannot therefore be attached to a hardware-only node, it needs to be able to execute software drivers.

PCI Express standard evloution

A step towards an easier device sharing, with virtualization support included, was made by the *PCI Express* standard itself [48, 46]. Two extensions were proposed: a *single-root* I/O virtualization (SR-IOV) and a *multi-root* (MR-IOV), the second being a superset of the first. They aim at multiplexing a device in hardware, so that it is perceived by a software stack as multiple devices. Registers of an SR-IOV compatible device are presented as one *physical function* (PF), a fully fledged device register, and multiple *virtual functions* (VF), of limited capabilities, but sufficient to use a properly initialized peripheral. By principle, the host (hypervisor) is supposed to initialize an SR-IOV device using the PF and then assign VFs to virtual machines (one VF per VM) to let them use it as if the device was assigned directly. A limitation of the SR-IOV is that it offers a hardware-supported device multiplexing on a single system node (being a *root complex*, in the standard terminology) only, which is not sufficient for a disaggregated architecture. The MR-IOV, instead, requires a dedicated switch connecting multiple *root-complex* nodes in order to enable the association of VFs with devices attached to remote nodes. Moreover, a PCIe chipset at each node has to support MR-IOV functionalities. The technology is supposed to serve well up to 16 or 32 *root-complex* nodes. Supporting more is not forbidden but the performance may be suboptimal [48]. It looks very promising although, reportedly, the availability of conforming hardware is very limited [61].

Tu et al.: *"Marlin: a memory-based rack area network"*

One PCIe based system by Tu et al. [57] enables sharing SR-IOV devices (specifically NICs) across disaggregated architecture. However, in this case the system architecture is slightly different. It is composed of one PCIe switch with the shared NICs attached to it together with the management host (connected through the *transparent bridge* (TB) port) and several compute hosts (connected through *non-transparent bridge* (NTB) ports). The management host is responsible for NICs enumeration and for the programming of translation tables at NTB ports, also at device discovery time. Each machine can get assigned one or multiple VFs associated to a given NIC and the NTB port is responsible for performing data transfers.

As in the case of all hardware-based solutions of a star architecture, the scalability of the system is determined by the capabilities of the central node, here specifically a PCIe switch. Thanks to the device enumeration and mapping to compute nodes, performed by the management host, no specific software negotiation phase is needed, as in the work by Hou et al. [30].

Suzuki et al.: *"Multi-root share of single-root I/O virtualization (SR-IOV) compliant PCI Express device"*

Facing the limited availability of a commodity MR-IOV-compliant hardware, Suzuki et al. [54] described a prototype system that attempts to achieve similar functionality of sharing SR-IOV endpoints (devices) among multiple hosts using custom components. The system does not require any modifications at a device side or its driver (differently from a standard MR-IOV approach) at the cost of lower performance. A virtual distributed PCIe switch (with a single virtual PCIe tree) is constructed over an Ethernet-based interconnect. In order to attach a node to the interconnect, two types of custom adapters are used, responsible for tunneling PCIe packets over Ethernet: a downstream bridge — attached to a device — and an upstream bridge — attached to a host. Each downstream bridge is identified by a VLAN number and a separate system manager host reconfigures the network for proper device assignment.

In one presented variant, with a single VLAN number per host, a peripheral can be

shared with other hosts only by taking turns. Before each assignment an update of the current configuration is required. Still, within a single machine, all hosted VMs could access the device simultaneously through a dedicated VF. Therefore, referring to [30], this kind of sharing could be sufficient for a GPGPU or devices that are not necessarily supposed to serve multiple hosts at the same time. However, this sharing scheme may not work well for peripherals that are routinely interleaving inputs from multiple hosts, like a NIC. If all VMs of a given host do not generate enough traffic, a NIC will remain underutilized. On the other hand, frequent reconfigurations may generate by itself such a significant overhead that the device utilization level will also be suboptimal.
In another variant, peripherals can be shared simultaneously between multiple hosts at the cost of each host having only one VF attached. Therefore, an additional device multiplexing between VMs has to be performed in software, which certainly would be marked by lower performance.

LegoOS and *dReDBox*

From the device disaggregation perspective, both the *LegoOS* by Shan et al. as well as the dReDBox design are very similar as they provide a peripheral attached to a *device node* (refer to figure 1-2) to the workload being executed on a *computing node* [53, 35]. In both cases, no specific requirements related to a device type is stated, the mechanism is supposed to support a generic device. Similarly, in both works there is assumed a custom hardware support and integration at the global system management level in order to perform steps like a device instance selection, interconnect configuration and mapping device registers.
Nevertheless, the difference is that in this dissertation there is addressed the subject of *direct-attachment* of a disaggregated device to a VM. In Chapter 5, there is explained why disaggregation broken the traditional way of configuring devices in this way and how it can be re-enabled again.

Summary

Plugging devices to a PCIe bus is the most common way of attachment nowadays and its recent extensions introduce hardware-based device multiplexing facilities. From

the virtualization point of view, it is especially important for the device *pass-through* attachment mode, in which a device is assigned exclusively to a single VM. The VM can use it directly, without exits to hypervisor. Logically (in terms of device drivers assignment), the device is in fact detached from the host OS. Such a configuration offers a near-native performance but disables the sharing of a device with other VMs.
The SR-IOV extension of the PCI standard addresses that problem but only for devices attached to a single *root-complex*. In a disaggregated system, it is highly desirable to be able to configure any device (or associated VF) in a pass-through mode, regardless of the actual node it is attached to. The MR-IOV is supposed to bring such functionality, also performed in hardware, but currently its adoption is very limited.
In Chapter 5, we present the design of a device sharing framework supporting architecture disaggregation from the ground up. Up to the author's best knowledge, systems with both disaggregated memory and devices (as depicted on figure 1-2) at the same time are not described by any position of the recent literature. The principle of this design is that a VM executed on a compute node uses only a disaggregated memory coming from a memory node. This VM is capable of configuring a peripheral (or a VF, depending of available hardware capabilities) attached to a device node in a pass-through mode in such a way that the peripheral may operate directly on the input and output buffers residing in a disaggregated RAM.

Chapter 3

Guest memory provisioning in a disaggregated system

3.1 Chapter introduction

Previous chapters introduced the context of this dissertation and discussed related positions from recent literature. In this chapter, we present the first main contribution, which is the design of a virtualization layer, enhanced with respect to memory provisioning.

A workload deployed in a *Virtual Machine* (VM) is capable of using only a certain fraction of system resources, accordingly to respective VM parameters. They are initially specified when a VM is started but it is also strongly desired that they can be dynamically adjusted at runtime for a flexible resource balancing. As explained in Section 1.3, the virtualization layer is essential for a data-center system and therefore the primary goal of all design choices and introduced components is to support it. The enhancements presented in this chapter provide the functionality of a runtime adjustment of the guest RAM with a strong emphasis on an easy integration with a disaggregated architecture.

Different ways of guest memory balancing were discussed in the Section 2.2, especially the part presenting an approach adopted in this work discusses rationales behind the memory provisioning approach. Regarding the presentation structure, subjectively it made the most sense for the author to present a complete picture of the design upfront and provide justification or alternatives discussion afterwards.

3.2 Proposed system architecture

In the design presented in this dissertation, when a VM starts, the guest's virtual RAM buffers are not reserved using the default host memory allocator (e.g. using the `malloc()` function). Instead, they are obtained from a custom host driver managing isolated ranges of *Host Physical Address space* (HPA). On a disaggregated system, resources coming from remote memory nodes are supposed to be mapped to these ranges at host level. Because of that, each guest memory section is also contiguous in HPA, as well as each isolated range is used by only one VM maximum (except for explicit memory sharing). Eventually, guest's virtual RAM is constructed of one or multiple such isolated chunks, which amount can be adjusted at runtime. This approach makes the virtualization layer easy to integrate with the architecture disaggregation.

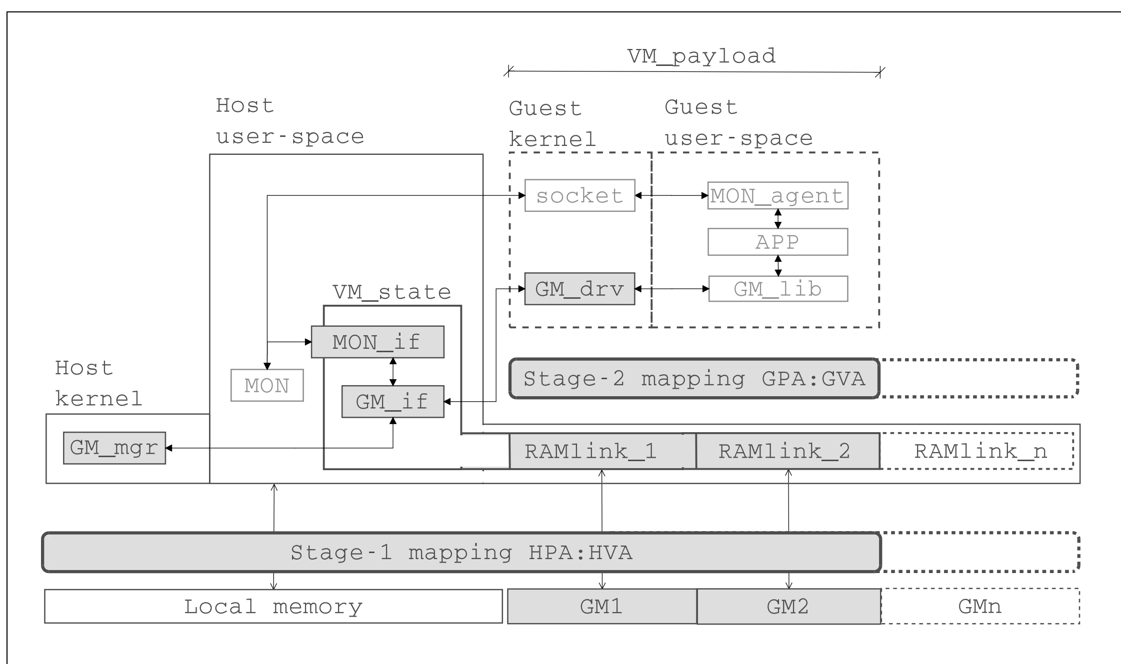


Figure 3-1: Guest memory provisioning from isolated pool

The related enhancements of the virtualization framework span three levels. In order to simplify their relative positioning, it is useful to have a look on the figure 3-1, although it will be better described in the further part of this chapter.

The first ehnancement is a custom host kernel driver that allocates the proper data structures related to the isolated memory. A description of this resource has to be given to the system at boot time in order to initialize the underlying hardware properly. For example, in the *dReDBox* prototype we used, this is done with the *device-tree*, a

description of the underlying hardware platform, typically used to inform the *Linux* kernel about the available resources and their parameters. Alternatively, this could be signaled by other components of the software stack which are executed on a platform next to the OS and capable of discovering the hardware topology (for example a UEFI firmware implementation [17]). The initialization process of the isolated RAM (representing the disaggregated memory) is very similar to that of the regular memory, but the corresponding ranges of HPA are not eventually passed to be managed by the system allocator (for example, the *buddy allocator* in *Linux*). In this way, they are not available for regular user-space processes running on the host. Instead, this memory is available solely for VMs.

The second level affected by the design-specific customizations is a software layer, depicted as a `VM_state`, which creates the *Virtual Machine* (VM) itself. As in a conventional approach, it is responsible for proper initialization of **guest memory**, which addresses undergo a two-stages translation (from *Guest Virtual Address space* (GVA) to *Guest Physical Address space* (GPA) and then from GPA to *Host Physical Address space* (HPA)), unlike for the memory allocated for a typical user-space process, where there is only one translation stage (*Host Virtual Address space* (HVA) to HPA). In our architecture each VM instance is spawned in the host local RAM, like for regular processes, but all memory buffers obtained to build up the RAM of the guest are allocated from the isolated memory pool. Moreover, while a guest is running, a VM is capable of expanding and shrinking the volume of the virtualized RAM. This operation is called the *guest-physical* memory resize. Finally, a VM interacts with the guest OS to receive resize requests and to coordinate the respective *logical memory resize* at guest side.

The guest OS is the third modified level, presented as part of the `VM_payload` on the figure 3-1. The guest exchanges messages with the VM and, based on the parameters it receives about the *guest-physical* memory chunks, it adapts the *logical memory* by performing *hot-add/-remove* operations. This includes the adaptation of the memory-related data structures that guarantee that the underlying physical resources are used in a correct manner.

A detailed description of design components

Figure 3-1 shows a detailed view of the components of the virtualization framework. The core components and their roles are described below. The less relevant ones are grayed out as they are not crucial for the mechanism itself but serve as a context illustration for the following sections of this chapter.

`GM`k**:** Guest memory *backends* — chunks of physically contiguous memory isolated from the host allocator and used exclusively to build guest RAM, $k \in [1, 2, \ldots, n]$. On a disaggregated architecture they represent resources from remote memory nodes.

`RAMlink_`k**:** VM-internal data structures abstracting guest RAM resources. Mapped *1-to-1* to `GM`k, $k \in [1, 2, \ldots, n]$, during initialization by the host driver `GM_mgr`. From a guest OS perspective they are considered as physical RAM.

`GM_mgr`**:** Guest memory manager implemented as a host driver. Upon request from a VM, selects a `GM`k region and maps it to the VM process address space, specifically to the `RAMlink_k` range.

`VM_state`**:** A set of data structures describing a guest system execution management and providing an auxiliary VM functionalities.

`VM_payload`**:** An actual memory range where the guest OS code has been loaded and is executed by a VM. Its size can vary over time but is upper bounded to the amount of memory provided by all *backends* attached to this VM at a given moment.

`MON_if`**:** VM component exposing an interface for external processes that could perform various management operations, for example stop a VM, resume it, query its parameters or trigger reconfiguration (e.g. memory resize). The interface can be, for example, a *telnet* server.

`GM_drv`**:** Guest-side driver, responsible for the communication with the VM and triggering the *logical memory resize*.

GM_if: A VM communication interface. In order to initiate the memory resize and exchange corresponding messages it interacts with several other components:

- Guest driver GM_drv; to exchange addresses of added/removed memory sections and corresponding signal messages.
- VM monitor interface (MON_if) to receive memory resize request (alternative way) signaled by external processes.
- Host driver (GM_mgr), to requests the binding of a RAMlink_k instance to an actual host memory chunk GMk.

The manager module (GM_mgr) is responsible for selecting a range of isolated memory (GMk) that will be linked to a given VM by a RAMlink_l. There are two cases to consider: the GMk to RAMlink_l linkage happens either when a new VM is launched or at runtime, when its RAM volume is resized.

- The first case is the most straightforward of the two because resources are properly initialized at host side and from the guest's perspective they are statically predefined, for example by the provided *device-tree*.

  The guest memory initialization takes place when a VM process builds a machine abstraction and allocates buffers that will constitute its RAM. Instead of using standard means (for example the malloc() call) to obtain resources from the host memory allocator, a VM (GM_if) uses a specific driver (GM_mgr) to reserve one or multiple *backends* (GMk). In this way, the guest RAM is comprised of one or multiple chunks of isolated RAM.

- In the second case (dynamic memory resize) additional steps are required. A VM has to receive a reconfiguration request in order to modify its current setup, so messages exchange is necessary between a VM communication interface (GM_if) and either a VM monitor interface (MON_if) or a proper guest driver (GM_drv); these are two different usage variants and they are discussed in sections 3.4 and 3.5, respectively.

  Additionally, respective runtime modifications are needed at guest side in order to make new memory resources available for guest processes, or, symmetrically, to

properly stop using the resources that are about to be detached. In Linux parlance, the former and latter are called *memory hot-add* and *hot-remove*, respectively. They modify the amount of guest memory at *memory section* granularity. A *memory section* is a physically contiguous range of memory. Its size is platform-dependent (for example 1 GB).

Whether it is a guest or a VM, that performs the reconfiguration first, depends on the type of operation. In the event of guest RAM expansion, new *guest-physical* memory resources have to become available before the guest can start using them. Therefore, the reconfiguration is done at VM side first, before notifying the guest. Symmetrically, in case of RAM shrinking, the guest has to stop using the involved *sections* first, before a VM can safely release the corresponding memory *backends*. Figure 3-2 presents a flow diagram of these operations in both cases, assuming they are triggered by an external process that sends a resize request to a VM through the monitor interface `MON_if`.

As a side note, when looking on the figure 3-1, it may seem that `GM_if` and `MON_if` could as well be merged into one module. Although that could simplify the presentation, it would obscure the functional separation. The former is specific to the presented memory resize method, whereas the latter encapsulates the communication logic in general. It can receive requests for different operations and thus interact with many other VM submodules, including, but not limited to `GM_if`.

3.3 Resize volume

The resize request received by a VM specifies a desired amount of memory, which, after comparing it against the currently available amount, allows to determine whether the guest's RAM should be extended (positive difference) or shrunk (negative difference). The absolute value of the computed difference must be converted to a multiple of a backend's size. In case of expansion, the safe assumption (from the guest workload perspective) is that **at least** the requested amount of RAM should be provided, therefore the absolute value of the difference is rounded up. Conversely, when shrinking the memory, one can assume that the workload is ready to execute with **no less than**

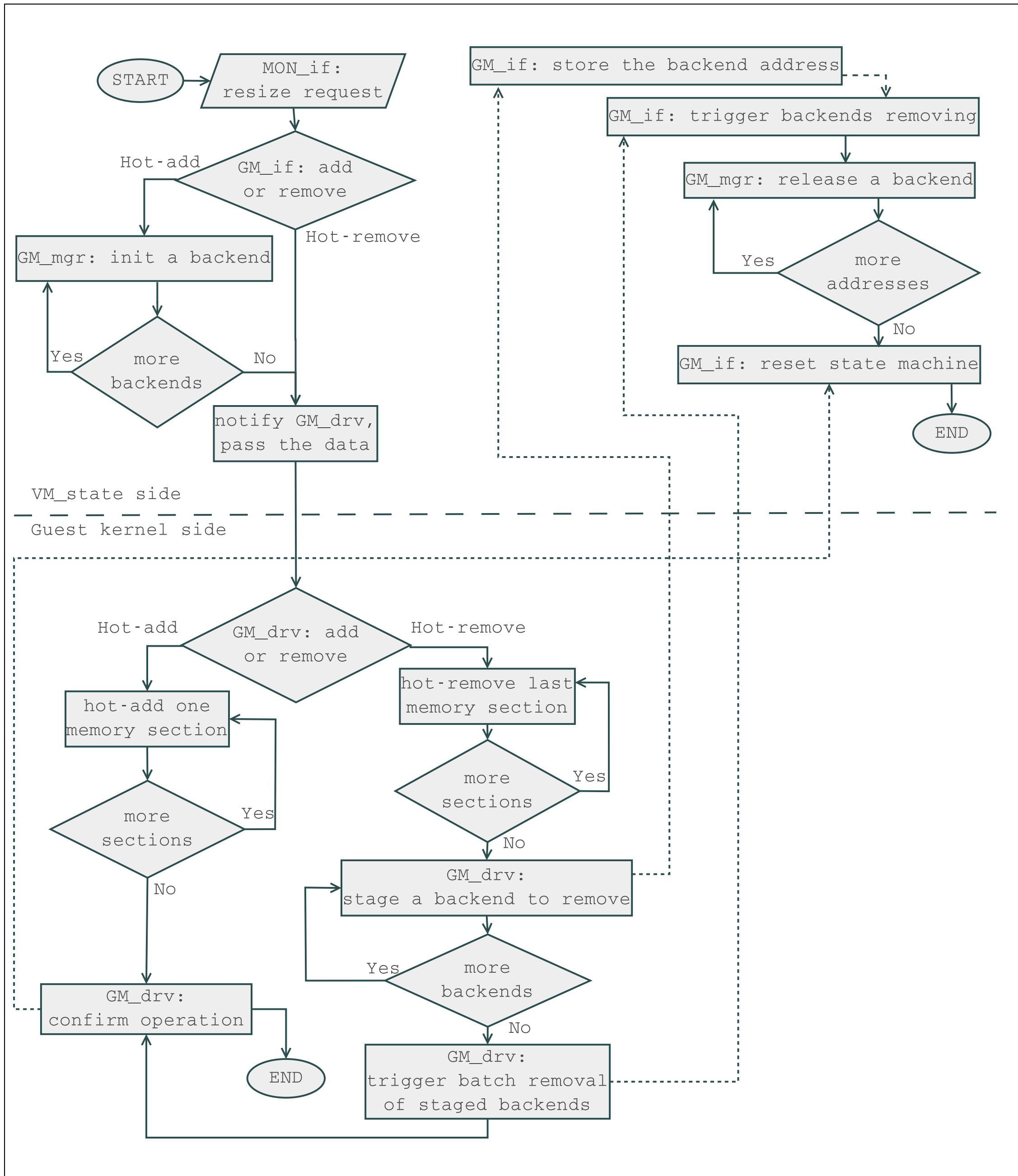


Figure 3-2: Dynamic memory resize

indicated amount of RAM, thus the absolute value of the difference is rounded down. Since each initialized *section* has to be associated to a VM *backend*, the size of a backend should equal to a multiple of the size of a *section* (possibly one). Additionally, the smaller the VM *backend* size, the better the granularity of the RAM resizing. Therefore, a VM *backend* size equal to the size of the guest *memory section* seems optimal.

3.4 Live VM balancing: guest parameters visibility

The diagram on figure 3-2 presents a situation in which a resize request is emitted through the `MON_if` component. Referring to figure 3-1, `MON_if` exposes an external interface, for example used by a monitoring software (`MON`) responsible for managing all hosted VMs and balancing the available resources between them. Such a software could make an arbitrary decision about changing the amount of a VM's RAM based on internal logic and usage data. Data could be periodically collected either from each VM process (through the same `MON_if` interface) or directly from the running guests, for example through an independent path, like a `socket`.

Observing only a `VM_state` (as opposed to considering also guest-internal parameters) has the advantage of no requirements towards the software installed on the guest but can only provide memory consumption data of limited accuracy. For example, a VM can possibly monitor the access frequency to *backends* (assuming that the monitoring overhead is acceptable) but it is hard to derive from that whether a backend can be reclaimed with no or negligible impact on the guest's performance. It could be that it has not been accessed for a period of time but is still needed. At most, it could be a good candidate for swapping the data to disk in case of high memory pressure on the host. However, on a disaggregated system, swapping should ideally not be needed. At least the process should store pages in remote memory instead of disk. Moreover, in order to reduce the amount of I/O operations, the *Linux* kernel leverages on unused pages and caches open files' content there. This mechanism is known as the *page cache* and the memory it uses is indistinguishable from regular data when observed from the host's perspective. If not limited by a specific system parameter, it can consume almost

all unused pages and thus create an impression that the guest is about to run out of memory, since cached files are not evicted until it is necessary. From the point of view of a VM the memory consumption of the guest is thus not a reliable indicator that the guest's memory should be expanded.

More precise data can be collected directly from a running guest, however this requires a dedicated service (depicted as `MON_agent` on figure 3-1) to maintain the communication and query proper parameters. Such a service could be running as an independent *daemon* process, alternatively accompanied by a supplementary guest *kernel module* if some required information could not be obtained with standard user-space tools. Although this would require guest OS modification, the components can be provided for a one-time installation, which is most likely acceptable. In this scenario, the host VM manager (`MON`) could collect more precise data (e.g. regular memory vs. *page cache* consumption) and resize the guest's RAM when really needed.

3.5 Explicit resize requests

Orthogonally to live VMs balancing, a RAM resize request could also be emitted by the guest itself, with two different possible strategies: *pro-active* and *reactive*, with respect to the moment when the guest runs out of memory, as illustrated on figure 3-3.

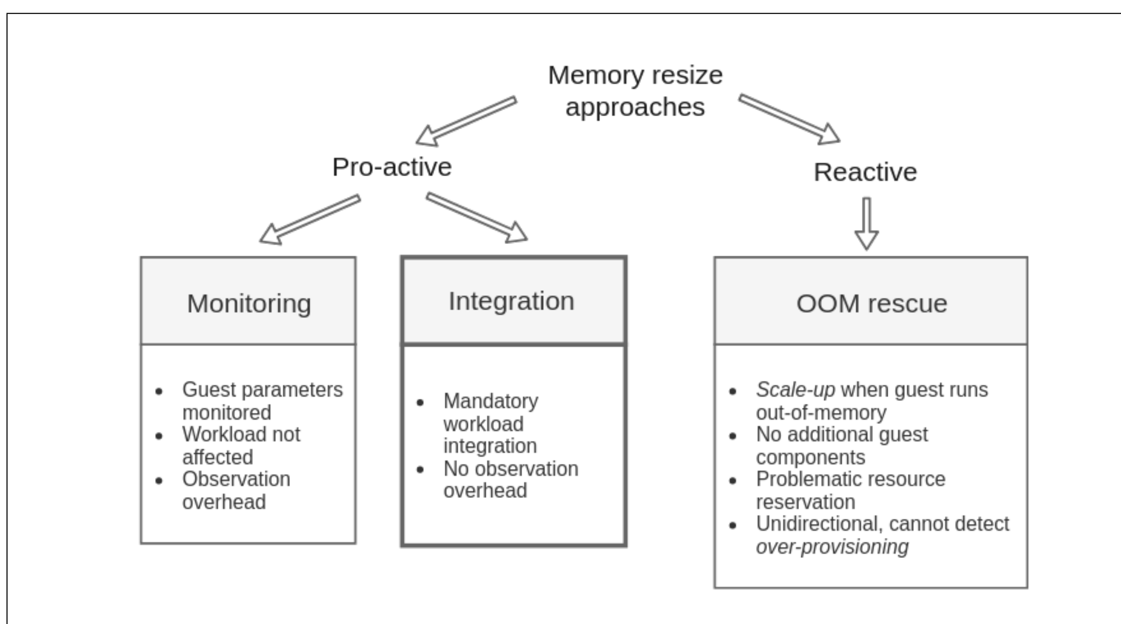


Figure 3-3: Different moments of issuing memory resize request

The *pro-active* strategy assumes that this situation could be avoided since the need for additional RAM can be anticipated. It can itself be split in two different scenarios:

- In the first scenario, a guest component (`MON_agent`) performs continuous monitoring of the relevant parameters and emits a request in case some of them (singularly or in a combination) cross over a given threshold. This scenario does not significantly involve a workload owner, which is good, especially for legacy applications that cannot be adapted to the system. Also, the monitoring logic could detect cases when there are not enough or too many sections attached and could thus handle both memory *under-provisioning* and *over-provisioning*. The drawback of this approach is that a continuous monitoring could imply a significant performance overhead, depending on the number of observed parameters [62, 20].
- In the second scenario, instead of being a standalone guest component, the `MON_agent` would be integrated with a user workload (`APP` on the figure 3-1), for example as a shared library. This requires that an application can notify the environment in advance about its need for more memory as well as it should cooperatively signal when the amount of attached RAM can be decreased in order to make them available for other VMs. This approach is much more precise and lightweight as the application knows exactly when and how much more resources will be needed. A continuous monitoring is expendable in this case at the price of an application logic being well integrated with a resize library that is an additional engineering effort at user's side.

The *reactive* strategy is conceptually the most straightforward, as no guest monitoring or application integration is performed. Instead, the guest is allowed to run *out-of-memory* (OOM), in which case it will ask for more resources before handling the situation by taking standard steps like page swapping or selecting a process to be killed in order to reclaim its memory. One potential drawback is that the process of RAM expanding needs a certain amount of memory itself in order to initialize the kernel data structures describing the new section, whereas the system just exceeded all its resources. Therefore, there is a need for a buffer, reserved *a priori* to serve such situations. There

should be at least one temporary buffer reserved for such *metadata* of a section attached in OOM state. However after the operation succeeds, its contents should be moved elsewhere, to allow for the next resize. Specifically, it cannot be moved to the newly attached section because it would render it impossible to detach later. Optimally, some data should be rearranged so that the *metadata* of all sections would be stored in the first one. This would allow all other ones to be removed at runtime, which gives the maximum flexibility possible.

Another drawback is that this strategy can only attach more resources and does not help with detecting sections that could be potentially detached. Therefore, the *reactive* approach seems more like a last resort solution in case *pro-active* techniques would not be applicable or sufficient.

3.6 Request path

The last element of figure 3-1 that we did not discuss yet is the `GM_lib`, exemplifying an alternative communication path between a workload and the host environment. Instead of using an independent channel to trigger memory resize, this component interacts directly with the `GM_drv` guest driver, the same that is responsible for passing the parameters between guest and VM during operations. This can be useful, for example, for guests that do not support networking or in a situation where there is no monitoring software running on the host, capable of receiving notifications over the network. Regarding figure 3-2, the execution flow is almost the same, with the only difference that `MON_if` is replaced by `GM_drv`.

Considering the presented framework as a *paravirtualized guest memory manager*, `GM_if` can be considered as a back-end while the `GM_drv` plays the role of a front-end.

3.7 Resize granularity

The presented system provides a *section-based* technique for dynamic memory balancing between VMs running on the same host. This is different from traditional *page-based* mechanisms like memory *ballooning* [59], which operates within the scope of the VM memory defined at boot time. As discussed in Section 2.2, our approach aims

to keep the runtime flexibility and to avoid the drawbacks of the *page-based* methods. The resize is based on memory section granularity in order not to introduce additional host's memory fragmentation and not to be constrained by a maximum value declared at launch-time. The price of this approach may be a lower consolidation level since fractions of sections from several VMs cannot be efficiently reused. However, *page-based* approaches are known to increase system fragmentation, which can effectively cause similar effects when the system having multiple tiny pieces of free memory will not be able to use them to allocate a larger object. Additionally, assuming that a typical cloud server machine is equipped with hundreds of gigabytes of RAM, a section granularity seems to be acceptable: its size is platform-dependent but typically not more than 1GB. Moreover, since applications' requirements are usually changing over time, operating at a coarser granularity (section instead of page) limits the volume of control signals traffic between host and VMs, otherwise required to pass a page ownership to another guest.

3.8 Disaggregation context

A re-sizable memory virtualization layer based on isolated *backends*, as described above, is ready to be adapted for disaggregated architectures. The `GM`*k* *backends* need to be associated with actual memory resources by the `GM_mgr` module at the host level but for a VM it is completely transparent whether they are backed up by local or remote RAM. Considering the fact that the amount of locally available RAM would no longer be a limit and the *memory-to-CPU* ratio per VM would be flexible, the granularity of a section size seems to be even more reasonable, as the amount of globally available memory may be in the range of terabytes.

3.9 Guest memory isolation

As mentioned before, the memory resources exposed as `GM`*k* *backends* are not handed to the host *memory allocator* but managed exclusively by `GM_mgr` instead, which always knows whether a backend has been mapped to a VM or is unmapped. Thanks to that, an unmapped backend can be detached from the host. This allows for a clean host

memory regulation, which can help optimal resources arrangement on a disaggregated system. Without this strict separation, all physical memory would be considered as a *local memory* on figure 3-1. Buffers of multiple *guest-physical memory* could then be interleaved with other host processes and the host memory fragmentation could increase over time. This, in effect, could result in one VM using multiple smaller buffers from several host *memory sections*, rendering them nondetachable, which could eventually lead to lower flexibility of resources arrangement in a disaggregated system.

3.10 Chapter conclusion

In this chapter we proposed the design of virtualization layer enhancements that can be easily integrated with disaggregated systems and support flexible guest memory resize at runtime. In Chapter 4 we will build on that in order to support also memory sharing between VMs and VM migration.

Chapter 4

VM memory sharing and migration

4.1 Chapter introduction

Building on top of the design from Chapter 3, in this chapter we present further enhancements of the virtualization layer taking full advantage of the resources provided from disaggregated memory pool. The design retains a full compatibility with runtime guest memory resize. However, the main subjects of this chapter are memory sharing between different VMs running across the system, a lightweight VM migration between different compute nodes as well as transparent VM data migration between two memory nodes. These two subjects of memory sharing and migration make the second main contribution of this dissertation.

As explained in Section 1.5, thanks to more a flexible hardware arrangement, resource scheduling can be much more efficient on a disaggregated architecture. In particular, a higher utilization level can be achieved with runtime workloads balancing. Ideally, the distribution of all deployed VMs can be periodically re-configured, such that only a minimum number of necessary computing nodes is active and the other ones remain powered-off.

Another important aspect of virtualization is memory sharing. In a disaggregated system shared regions are also coming from remote memory nodes and therefore they can be used by different VMs, not necessarily co-located on the same computing node, with no additional data copied by software means. The access is performed similarly to a regular memory region, only with additional synchronization steps. Most importantly,

there is no need to switch to network-based communication in case of VMs launched at different computing nodes, as in some works mentioned in Section 2.4.

4.2 Memory sharing — overview

This section is related to the concept of inter-VM disaggregated memory sharing and corresponding access serialization, both based on a POSIX-like API, thus easy to adopt by existing applications.

Just like with classic clustered data centers, there are two different situations to consider: VMs may be running on the same computing node or different ones.

In the first case, provided that enough memory is available, the shared memory could come from the local RAM (similarly to IVSHMEM [12]). This would perhaps result in a lower access latency, compared to memory of a different node. On the other hand, this would increase the amount of data to be copied in case of a `VM_state` migration (described below). Moreover, it prevents VMs running on other computing nodes from joining the sharing group after it is created. In case of a disaggregated memory range being shared by VMs on the same CPU node, the *Global System Orchestrator* (GSO) still needs to be aware of that. It is indeed necessary, although it may seem to be avoidable, to register the fact that a certain memory region is accessed by multiple *Virtual Machine* (VM) so that it will not be detached from a compute node until the last user releases it.

In the second situation multiple computing nodes are attached to the same memory node and capable of accessing the same region(s) within. Because it affects multiple system nodes, such sharing needs to be established through the GSO.

The second case is more generic in the context of disaggregation and therefore it is discussed in details in the following. Regarding concurrent accesses to shared resources, a new software component the — the *Arbitration Unit* (AU) module — is introduced in this chapter. It is an abstract system module responsible for granting access rights to requesting workloads. Different possible locations of the module are compared, with respect to system scalability or required independent signaling channel between nodes. We also discuss the implications of implementing a strict *lock enforcement* mechanism

for shared accesses, instead of using locks as a programming convention only. Furthermore, different mechanisms of shared memory attachment are elaborated, taking into consideration the size of a region as well as the minimization of the proportion of HPA ranges of a given compute node that are reserved but not effectively used. Finally, we present a way to turn an already used region of regular memory into a shared one, with a zero-copy approach.

4.3 VM migration — overview

In this section, we describe an evolved notion of VM migration. Traditionally, VM migration meant taking a snapshot of guest memory together with the CPU and devices states, which usually represent altogether a large amount of data, and copying it from one machine to another. In the disaggregation context, a VM's memory footprint spans multiple different locations, the local CPU node and at least one memory node, which parts are described as `VM_state` and `VM_payload`, respectively. Both were already introduced in Section 3.4 and are also illustrated on figure 4-1. Thus, in the disaggregation context the term of migration gains a twofold meaning.

- **VM state migration**, consists in moving the `VM_state` data from the local RAM of the source (*outgoing*) computing node to the local RAM of the destination (*incoming*) computing node. This is probably the most intuitive meaning of the term *migration* because of the similarity with its meaning in conventional systems. The difference is that all guest memory buffers (`VM_payload`), representing the huge majority of the total volume of a VM's snapshot, come from disaggregated memory pool and they are not moved. Instead, the *incoming* computing node must be connected to these resources and make them available to a new VM process, that will receive the copy of the VM state from the *outgoing* CPU node. The interconnect reconfiguration is a responsibility of the GSO, which is also initiating the operation at the very beginning. In order to avoid data corruption, the *incoming* VM should start guest execution after the *outgoing* VM is stopped. Nevertheless, the expected downtime is relatively low as the amount of data to be copied is typically less than 1MB. Therefore, the `VM_state`

migration is a very lightweight operation which may open new opportunities for server consolidation improvements. The resource scheduler logic (introduced in Section 1.5) could take advantage of VM state migration more frequently for its tiny implied service downtime.

- **VM payload migration** consists in moving from one memory node to another only data residing in a disaggregated memory pool. The whole VM payload can be migrated or only a part of it, if it is spread across multiple memory nodes. Ideally, this should be almost transparent to the associated computing node, in the sense that there is no specific action required from the VM side. However, with the simplest *stop-migrate-resume* approach, the VM may experience longer memory access latencies along the process, because the volume of data to move is typically large (in the order of several gigabytes) and therefore, it is preferable to not withhold all memory accesses for the whole operation time. This is a major difference with the VM state migration. The operation cost could be reduced with hardware acceleration or by adopting *live migration* techniques ([24, 25, 29, 41, 45]). Also, an intermediate memory cache could help hiding this overhead.

4.4 Software modifications

Migration implies no guest kernel modification at all and this is because of the operation concept itself. It is not a guest's responsibility to decide whether it should be migrated (in any of the two ways discussed above) as the fact of running inside a VM is almost completely transparent to it. Except for some additional software components it may have installed, a guest is executing as if it was running on a physical hardware. The migration logic is divided only between the GSO and the VM. Ideally, the guest and processes it executes would not even notice the fact of being migrated, however, in practice, its execution may need to be stopped for the `VM_state` migration or its memory access latency may be longer while the `VM_payload` is being copied.
Regarding disaggregated memory sharing, in addition to the virtual device which is a VM component, an associated guest kernel driver is needed. This is because an interaction with the GSO is performed at host side. Therefore the driver only ex-

poses an interface to guest user-space workloads and delegates most of the work to the VM. Additionally, having a shared memory region successfully attached and reachable through a range of GPA addresses, the driver also maps it to the GVA of the requesting workload. Eventually, the driver is used to obtain (and release afterwards) a lock for proper serialization of concurrent accesses to shared resources.

4.5 Proposed system architecture

This chapter further expands the system architecture presented previously, which was related to provisioning guest memory as isolated sections coming from a disaggregated memory pool. Therefore, crucial components of the system design illustrated on figure 3-1 remain unchanged and they were repeated below in figure 4-1 for the convenience of the reader. However, several less relevant components were removed for the clarity of the presentation. The connectivities with the *Global System Orchestrator* (GSO) and the interconnect were emphasized.

The GSO is a heart of the systems, performing various management and configuration operations. In the context of memory disaggregation, it is the only component that can establish an interconnect link necessary to attach a range of remote memory to a computing node. Except for that, it also keeps a database of existing connections both for regular and shared memory regions. Finally, all that can be done on demand, through an interface used by computing nodes.

4.6 Sharing disaggregated memory

In the virtualization context, memory sharing means allowing multiple *Virtual Machine* (VM) to access the same range of physical RAM. With a disaggregated architecture, these VMs are, in the most generic case, located on different computing nodes and they need to access the same regions of one or multiple memory nodes. This situation is simply illustrated on figure 4-2, where two computing nodes have attached multiple sections from different memory nodes and mapped them to local HPA ranges. The GSO needs to uniquely enumerate all resources available across the system to be able to setup the interconnect paths accordingly. As introduced in Section 2.3, for this purpose we

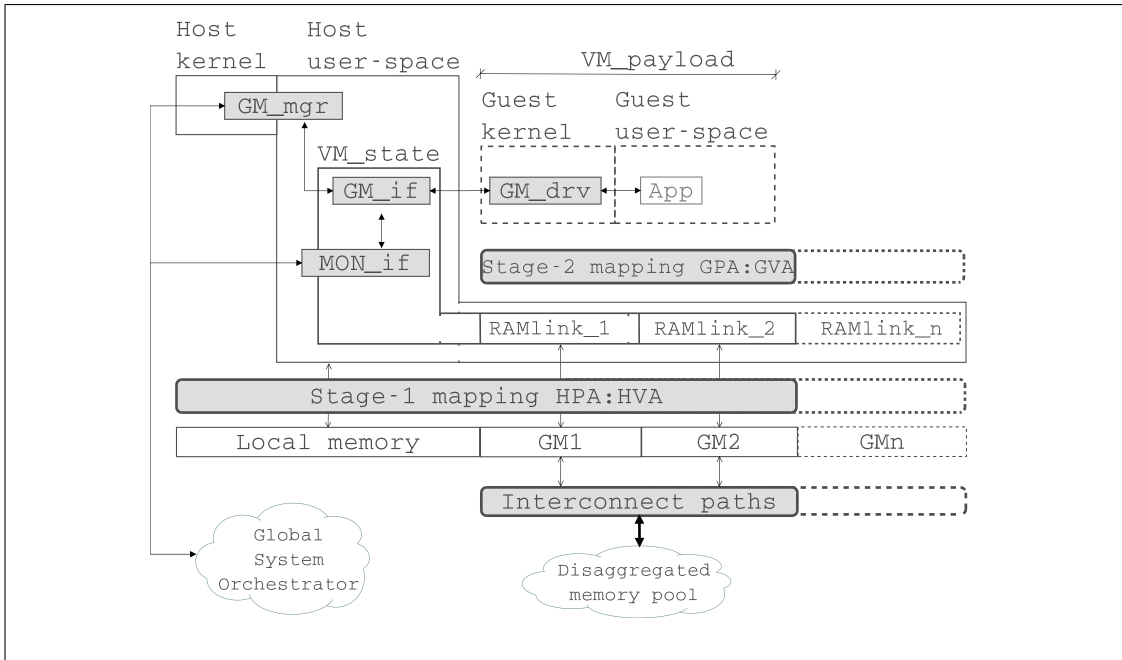


Figure 4-1: Virtualization framework components

adopted the *Uniform Physical Address space* (UPA) concept, describing each physical range of each system node.

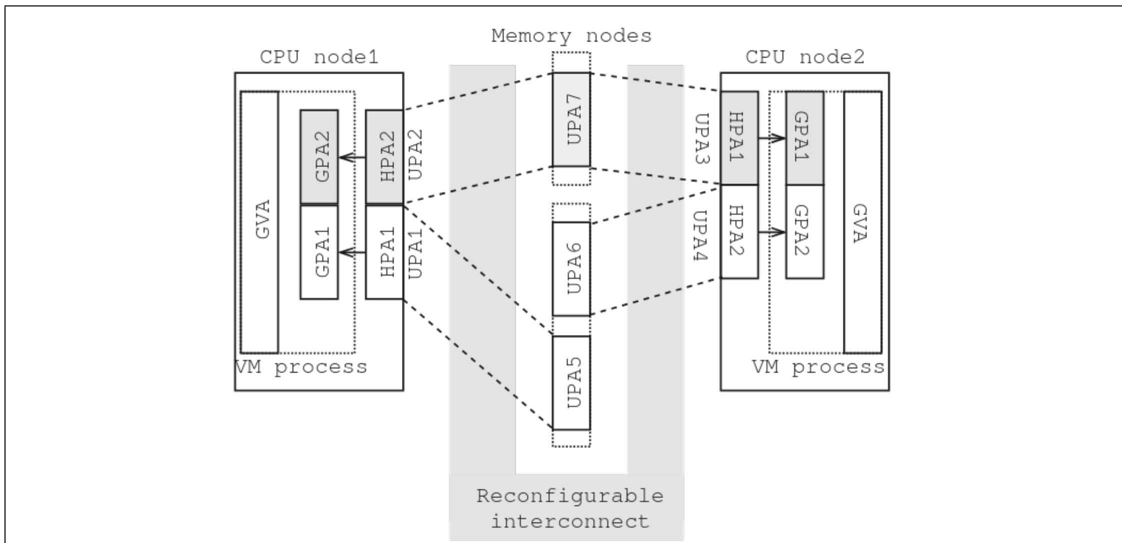


Figure 4-2: Sharing disaggregated memory

For the user space processes of the computing node, the attached memory ranges can be subsequently mapped to *Host Virtual Address space* (HVA). But the most important is that these ranges are also translated to *Guest Physical Address space* (GPA) of a given VM, and further to *Guest Virtual Address space* (GVA) of a guest kernel or guest user space processes. In this way, chunks of disaggregated memory are exposed as guest

RAM.

In figure 4-2, range `UPA7` is attached to ranges `UPA2` and `UPA3` of computing nodes 1 and 3, respectively. Subsequently, it is accessed by multiple VMs. Note that there could be more than 2 attached ranges and computing nodes. This is a simplified concept of disaggregated memory sharing and a careful reader will quickly realize that additional aspects have to be considered:

- Which component decides which ranges can be shared?
- How is the sharing established?
- How can the accesses be synchronized?
- How big can these ranges be?

These questions and a few other related questions are discussed in the following where we present the API of a library executed at the host user-space level and used by VM processes. However the guest is also equipped with its own paravirtualized version, such that guest workloads are using exactly the same APIs and the corresponding computations are conducted at the host side, with respective addresses translation along the way. For further reasoning, we assume that a VM has a way to derive an UPA address from the HPA it is passed (specific to a given computing node) and conversely.

Shared memory initialization

Similarly to well know approaches (like POSIX shared memory), in the proposed design we assume that a memory range is shared between a group of VMs and that one group is associated to one shared range, indicated by the *owner*. The latter, one per group, is a VM that invokes the initialization routine:

```
shmem_init(upa_addr_buff, size) -> shm_id
```

`shmem_init` takes two arguments, completely describing a contiguous range: a pointer to the buffer (`upa_addr_buf`) containing the UPA range address and a `size` of the range. The request is received at the GSO side, which registers a range as shareable and returns the sharing group identifier `shm_id` back to the caller. This identifier is globally unique and has to be used by other VMs wishing to join the group.

Additionally, we envision that, instead of turning already possessed memory into shared memory, a VM may wish to obtain a new region to be shared. In this case `upa_addr_buff` is set to `NULL` by the caller and the GSO fills it after selecting and attaching new resources to a given computing node. Together with the `shm_id`, this address is a second information passed back to a VM. It allows to correctly map the shared region to the guest. This situation is actually very close to what happens after joining a sharing group, as described in the next paragraph, with the difference that the VM becomes the group *owner* automatically.

Joining memory sharing group

In order to use a range of memory shared by the *owner*, an *accessor* sends a request to the GSO by calling the `shmem_join` routine:

```
shmem_join(shm_id, upa_addr_buff, size_buff) -> result
```

The routine indicates which region is to be joined (specified by `shm_id`) as well as it passes pointers to two buffers where the GSO writes the UPA address at which a new section was attached (`upa_addr_buff`) and its size `size_buff`. The returned `result` value indicates whether the values stored in these buffers are meaningful (on success) or not. Given that, a VM can map the shared memory region to the guest. The methods for the attachment of shared disaggregated memory are discussed in the next sections.

Up to now we silently assumed that all *accessors* have a way to discover the proper `shm_id`. By default it is generated by the GSO and returned to the *owner* when a sharing group is initialized. This is the easiest way because the `shm_id` is then guaranteed to be unique across the system. But of course, there is a need for an independent signaling channel in order to pass the identifier to all *accessors*.

For example, all VMs that are expected to share memory could be connected by the network and have their IP addresses coming from the same *VLAN*. It would then be quite straightforward to design a protocol for an *owner* discovery and `shm_id` exchange.

Otherwise, if VMs cannot be interconnected, they must possess a piece of *a priori* information in order to identify a sharing group. Still, we should avoid setting the

shm_id to a fixed value, in the sense that a VM knows all possible ones at boot time. To simply illustrate, all VMs running the same workload could be identified by a workload_id, known to them at VM boot time. The owner could provide the workload_id and an optional label string (both as additional API parameters) when requesting for a shared region initialization. This would allow the GSO to associate it with the newly generated shm_id. Then, the workload_id could be used by *accessors* to discover all initialized shared memory groups (thus shm_ids) relevant for a given application. Additionally, as a part of the discovery response, each shm_id could be as well accompanied by a label (like *"Input1"*, *"Output2"* etc.), meaningful for a given workload, so that a correct range can be easily figured out by the workload. This kind of approach no longer requires the VMs to have an independent signaling channel, at the cost of additional identifiers and, most importantly, of involving the GSO in the shm_id discovery. The latter is a central component of the system, so any additional processing load should be put on it carefully, otherwise it can affect the system scalability.

Leaving and destroying shared memory

Symmetrically to shared range initialization and joining, two routines allow to leave and destroy a shared memory region.

```
shmem_leave(shm_id)
```

is used by *accessors* to indicate the GSO that a mapping between a shared memory region and a given computing node can be torn down. At the same time, it is up to the VM to clean the corresponding mappings at the compute node side.

```
shmem_destroy(shm_id)
```

is invoked only by the shared region *owner*, to instruct the GSO to invalidate the shm_id, which renders the region no longer shareable. What happens if there are still *accessors* of the registered region is up to a particular implementation: the routine could return a failure status, the region could be forcefully detached, the behavior could depend on additional flags passed to the routine... Eventually, it is also up

to a particular implementation to decide whether the *owner* should remain capable of accessing the region as a regular memory or if the region should be detached afterwards. For example, the first option could be used when the shared memory was created from previously possessed RAM and the second option could be used when a new dedicated range was obtained by the *owner* for sharing purpose. Other implementation choices are possible.

Preventing data races

As with most resource sharing situations, concurrent accesses to the same range of disaggregated memory must be serialized to avoid data corruption. In this work we do not consider *time windowed mechanisms* where accesses are granted according some predefined scheme because they are usually tuned for a specific use-case. Instead, the presented approach fits better a generic data-center, which can host workloads of different type at the same time. It is therefore assumed that the members of the sharing group can perform accesses concurrently without any particular pattern. Like in well known solutions based on mutexes or semaphores, we propose an access locking mechanism based on a condition testing, performed atomically by the AU, introduced before in the Section 4.2

Two routines are called by competing VMs wishing to access the shared memory:

```
shmem_lock(shm_id) -> result
shmem_unlock(shm_id) -> result
```

`shmem_lock` is invoked to obtain an exclusive access to a shared memory region. It returns a code indicating whether the operation was successful or not. Symmetrically, `shmem_unlock` releases the lock. Differently from *mutexes* or *semaphores*, the above functions do not provide a generic locking mechanism to be used between different computing nodes, they are purpose-specific because of the `shm_id` argument.

The routine invocations are received by the AU module. There are several candidate locations for the AU module:

- As a part of the GSO — this may be the first intuitive idea as the GSO, at the heart of the system, is already involved in establishing the interconnect paths between

computing and memory nodes as well as it is connected to all system nodes. But the memory attachment or detachment is a relatively rare event, compared to the number of shared accesses that require serialization. Therefore, involving a central component of the system in the access rights alternation for all shared memories is probably a bad design choice: it would create a bottleneck and limit the system's scalability.

- As a part of the *owner* VM — this assumes that during the shared region initialization (`shmem_init`), the *owner* starts a dedicated AU service receiving lock requests and sending responses to a given sharing group. Provided that all *accessors* are connected to the *owner*, like for the `shm_id` discovery, this approach seems better than the previous one. It does not limit the system's scalability because the lock contention depends only on the number of members of the sharing group, not on the total number of members of all sharing groups. The main drawback is that handling a very high rate of incoming requests may take over the execution time. Other tasks running on the same computing node might be affected, including completely independent VMs (this kind of negative phenomenon is called *cross-VM interferences*).

- At a memory node — this technique requires that, instead of an *owner*, each memory node hosting at least one shared region runs the logic handling lock requests, as presented on figure 4-3 (similar to figure 4-2). All memory nodes with shared regions must be equipped with additional processing power. On the other hand, this may be a solution for scenarios where an *owner* and *accessors* VMs cannot be interconnected by an independent channel (like for `shm_id` discovery). When an *accessor* is joining a sharing group it must map the AU interface registers, together with the disaggregated memory ranges, and in the same manner. Moreover, since this is an external device, a corresponding interrupt must also be routed to the computing node and subsequently to the proper VM. We assume here that requesting a lock (that is, writing to a device register) is an atomic operation done by a VM, which is afterwards notified by the interrupt when exclusive access rights are granted.

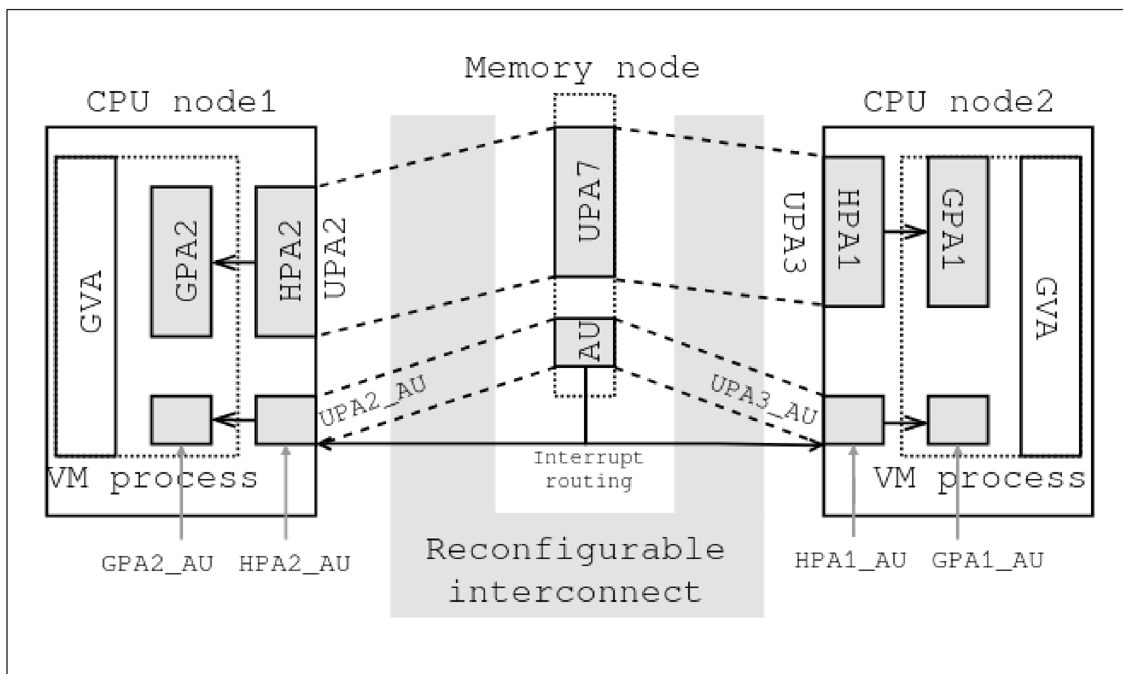


Figure 4-3: Mapping a disaggregated *arbitration unit* located at a memory node

Regardless of the method employed to obtain a lock, a general rule of thumb is that a shared memory region can be accessed only with a lock held. There are optimization techniques that can be build on top of this. For example, multiple *read-only* accesses could be allowed when no *writing* is performed at the same time.

The requirement of holding a lock is very often only a programming convention, which means that a careless user is still capable of accessing the shared memory without it, with data being possibly corrupted in an unpredictable way. Fortunately, the presented architecture allows to implement a lock-enforcement mechanism, for example as a 1-bit flag associated to the page address mapping. This flag could be set or unset by internal code of the `shmem_lock` and `shmem_unlock` routines, respectively, in order to allow an access only when the lock is granted. The enforcement could be implemented at the level of UPA–HPA translation, by a proper logic (which presence is presumed but not discussed here). Then, even inadvertent accesses performed from the host would be filtered. Alternatively, the lock-enforcement could be performed at higher level, by a VM process, with the flag embedded in the GPA–HPA mappings, traversed by the *Memory Management Unit* (MMU). Moving the mechanism to GVA–GPA translation tables makes less sense as these are managed by the guest OS and it is usually preferable to keep it as generic as possible.

Finally, this lock-enforcement provides an explicit protection against data races but at the cost of changing the flag for multiple entries (one per memory page by default) at each locking and unlocking operation (which may be very often). The number of modifications could be minimized if one flag would describe a contiguous block of the size of a minimal shareable region, and covering multiple subsequent pages instead of one. Nevertheless, most recently resolved page address translations are cached because they will be likely used again in a close future so the cached values can be reused. For example, inside a MMU, this is usually done using a *Translation Lookaside Buffer* (TLB). The additional cost of the lock-enforcement mechanism is related to the fact that, when a given translation map entry has one of its flags modified, its previously cached version has to be invalidated. The benefit of caching is lost in this case, which can significantly degrade the system's performance. For all these reasons the frequently preferred way consists in only using the locking as a programming convention.

Size of a shared region

Up to now we presented a simplified view where the whole section of disaggregated memory, attached to computing nodes, is shared (figures 4-2 and 4-3). But this is a very unlikely scenario because shared regions are typically smaller.

It is possible that only a subrange of a section could be mapped to the GVA of an *accessor* VM process, but the whole slot of a memory section would be occupied anyway. It means that the non-mapped part of the address space would be wasted, first at a computing node level (HPA) and then at a VM level (GPA) too. The reason is that both *physical* (HPA) and *guest-physical* (GPA) RAM is managed at section granularity. Moreover, it would be the guest OS responsibility to let the requesting workload access only the specific shareable subrange of the whole section. That is, again, because memory resize operates on a section granularity, the OS cannot attach only a fraction of the section. An access filtering in the guest introduces additional guest OS customizations (which should be avoided as much as possible) as well as it creates a risk that, in case of an exploited vulnerability, the workload could potentially gain an unauthorized access to a larger part of the shareable section than what is actually permitted. Since the whole section is attached anyway, it could stealthily reach the remote resources used,

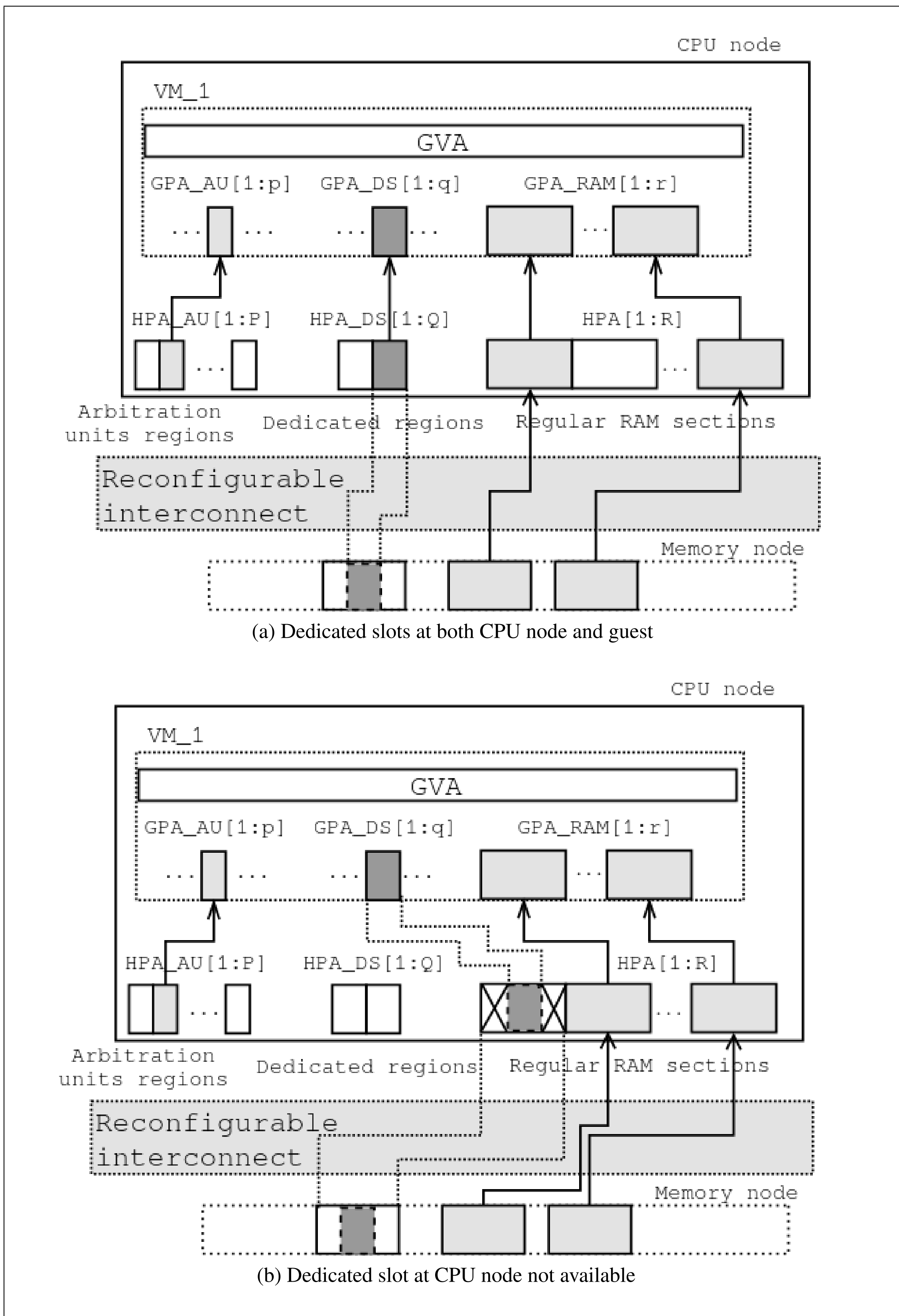
for example, by different sharing group of VMs.

Furthermore, each computing node can run multiple VMs with multiple sections from a disaggregated pool attached as regular guest RAM. Each VM may also want to access multiple shared regions of disaggregated memory and, in the worst-case, each one could imply attaching another separate section to the node. If this is the case, there is a possibility that a large number of shared regions (coming from different disaggregated memory sections) could prevent a section of regular RAM to be attached. This is because, in practice, the amount of supported memory slots (GMk on the figure 4-1) is limited by many assumptions existing in the operating system code. This means that a new VM could not be scheduled on the computing node, while several slots with shared memory regions are only partially used effectively. Therefore, a certain fraction of occupied memory slots would be wasted in this case.

In order to improve the utilization level of occupied HPA (and then GPA) ranges, we propose an additional way of attaching shared memory regions. For the purpose of this discussion, the RAM sections, both at HPA and GPA level, will be referred to as *regular* regions. In addition, the term "*dedicated* regions" is introduced to indicate smaller ranges to which a shared disaggregated memory is attached.

We propose an approach where a disaggregated shared memory is attached at the CPU node level (HPA) through either *regular* or *dedicated* slots, but, at the VM level (GPA), is only mapped to *dedicated* slots (except for the very specific case of the whole memory section being shareable, which will not be further discussed). Both *regular* and *dedicated* attachments are shown on figure 4-4 which depicts only the crucial changes from the already presented design.

In addition to non-shared RAM, mapped to *regular* slots, each computing node has also a reserved pool of *dedicated* slots of different sizes, meant to be used as a primary choice for attaching shared regions, as shown on figure 4-4a. Alternatively, if for any reason a *dedicated* slot of a desired size is not available on a computing node, a regular one can be used as a fallback solution, as depicted on figure 4-4b. However, the guest OS stays oblivious to that because, at VM level, the shared subregion would be still mapped to a *dedicated* slot. Thanks to this the access control logic is implemented in the VM code and at the HPA to UPA translation level. In both cases, there are also slots



(a) Dedicated slots at both CPU node and guest

(b) Dedicated slot at CPU node not available

Figure 4-4: Disaggregated shared memory attachment

for AUs mappings, used as described before for the synchronization of the accesses to both *dedicated* and *regular* shared memory regions.

We propose that the *dedicated* pool has several groups of slots of different sizes reserved in the physical memory layout of the computing node, with each group containing multiple slots of the same size. Group sizes should span a spectrum from a few kilobytes up to a major fraction of a memory section size (for example 75% of it). Larger shared regions would then use the approach of figure 4-4b. Up to author's best knowledge, there is no straight-forward formula, specifying how many slots of each size should be available for an optimal system performance. It depends on the total amount of memory supported and the demand for shared regions (thus, on the memory profile of deployed workloads). Therefore, we suggest that during the system calibration process related heuristics should be figured out, that make the system working fine. Building such a statistic memory model is out of scope this work and probably a complete research topic by itself.

Similarly, each VM should have a corresponding pool of reserved ranges of the exact same sizes but of lower quantity per each group because the pool of a computing node is supposed to be shared between all hosted VMs. There is only one VM on figure 4-4 but a computing node would typically run multiple ones. A VM would attach *dedicated* regions to the guest OS as *memory-like* devices at startup or dynamically at runtime, as if a device was plugged-in.

Referring to figure 4-1, a *dedicated* slot is attached on demand to a computing node by the GSO when the `GM_drv` of an *accessor* VM requests to join a sharing group (by calling `shmem_join`). Once received by the `GM_if`, the request is then passed to the `GM_mgr` component of the host, which is responsible for obtaining shared resources from the GSO and mapping them to the VM process (`VM_state`). On top of that the VM builds the *regular* or *dedicated* memory regions and exposes them to the guest.

As already mentioned, in order to share disaggregated memory regions, the *dedicated* slots of a computing node are preferable. The *regular* slots should only be used for shared memory as a fallback solution because they increase the amount of wasted HPA ranges. It is important that the slot selection is performed by the `GM_mgr` logic and that it is transparent to the guest OS, otherwise wrong configuration of one VM (for

example enforcing *regular* slots for all shared memory regions) could potentially affect the consolidation level of a given computing node.

As a matter of fact, different computing nodes might use *dedicated* host slots of different sizes (depending on the slots' availability at each node) in order to map the same shared range of disaggregated memory.

Additionally, multiple VMs hosted on the same computing node and participating to the same sharing group will use the same *dedicated* slot to access a given shared region (figure 4-5). Therefore, such a slot will not be released as long as there is at least one active user of it. The same applies when a shared memory is attached to the computing node as a *regular* section, as on figure 4-4b. The main rationale is to minimize the amount of HPA ranges that are occupied but not effectively used. In this particular case they could be shared with standard mechanisms oblivious to disaggregation (for example IVSHMEM [12]), however this would be incompatible with the assumed locking mechanism. Additionally, the GSO, if unaware of the established sharing, could decide to migrate either of participating VMs to another computing node without proper memory remapping, which would break its workload execution.

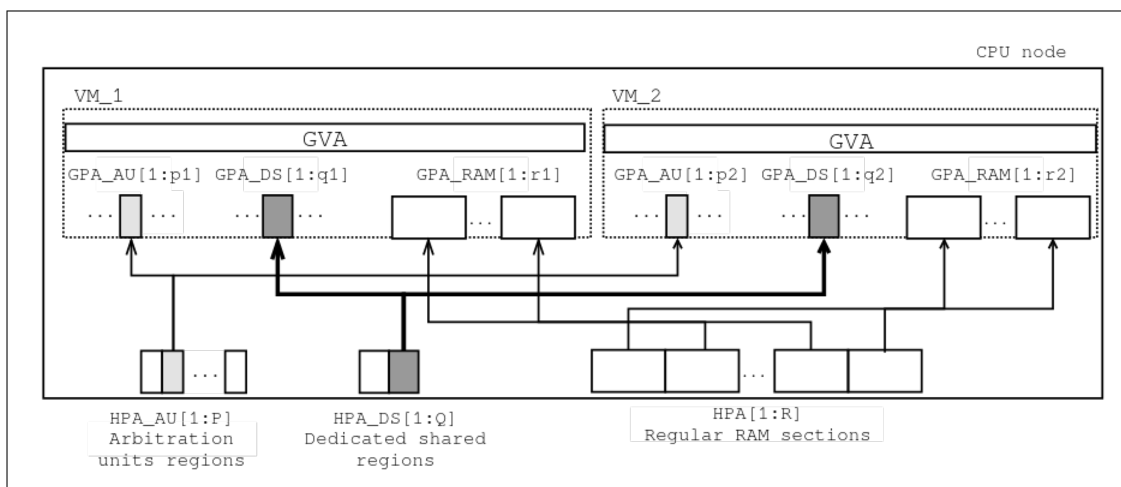


Figure 4-5: VMs of the same sharing group reusing dedicated slots

Initialization with existing data

Mapping of *dedicated* shared regions is always created on demand and it can be initialized by the *owner* (as mentioned at the beginning of in Section 4.5) with data from a non-shared guest RAM, that is a *regular* slot. For this purpose, the address of an

existing data buffer can be stored in the buffer passed to the `shmem_init` function, to request the GSO for a new *dedicated* region initialization.
This case is illustrated by figure 4-6. When a subrange of a *regular* guest RAM is about to be shared, it will be remapped at the interconnect level to a *dedicated* shared region of the same computing node but no data needs to be copied. An important step is that

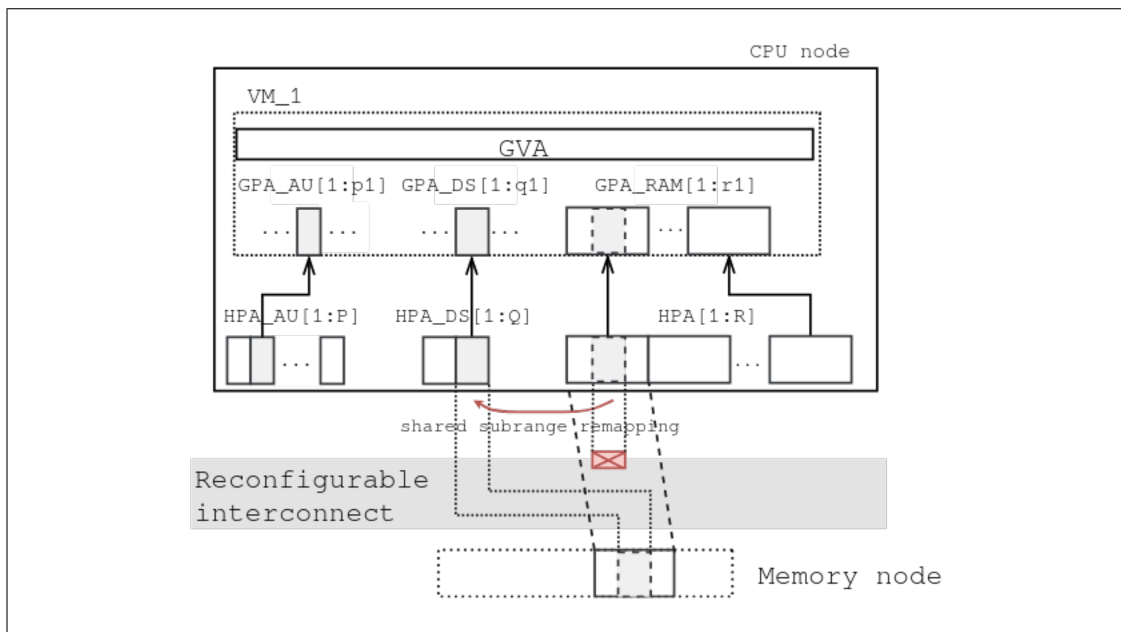


Figure 4-6: Dedicated slot initialization from an existing RAM

the previously used mapping is marked as unavailable. It should not be completely erased, however, as the region can be turned back to a non-shared one at a further stage of execution, but the point is that a shared disaggregated memory region can only be accessed through pointers obtained through the `shmem_init` or `shmem_join` routines. Only these pointers are compatible with the locking mechanism described above and guarantee correct data access serialization.
In other cases where no *regular* RAM range is indicated at shared region initialization, a *dedicated* region will be mapped to a newly allocated range coming from any memory node, selected by the GSO.

4.7 VM migration

As mentioned in the Section 4.3, the term of *VM migration* became ambiguous in the disaggregation context, as there can be distinguished *VM_state* migration and *VM_payload*

migration. The figure 4-1 presents the difference between those two in terms of system components involved, while this section is focused on an actual data location and discusses how both types of migration are moving the data and reconfigure the interconnect accordingly.

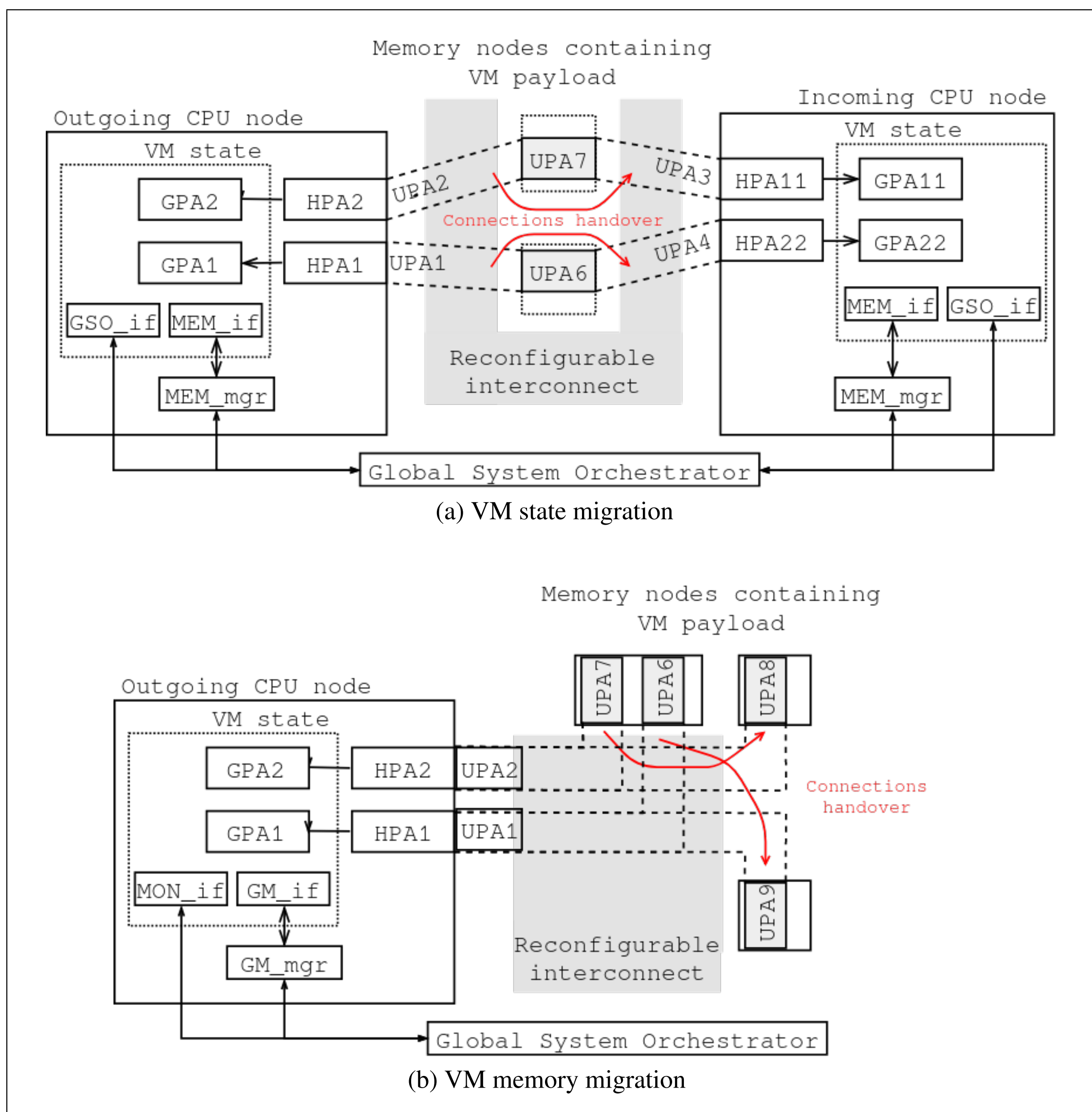


Figure 4-7: Two meanings of a disaggregated VM migration

VM state migration

Migrating the `VM_state` means moving the part of the VM's memory footprint residing in the local RAM to another computing node. The migrated data describe the current state of the CPU, device registers, list of devices configured in a *pass-through* mode (physical devices used directly by guest drivers) as well as links to buffers from

disaggregated memory that are building up the *guest physical* RAM... Therefore, the VM_state can be seen as a set of *guest metadata* describing the current state of its execution environment.
The VM_state migration process is conducted by cooperation between the *outgoing* VM, the *Global System Orchestrator* (GSO) and the *incoming* VM as illustrated on figure 4-7a.

1. The GSO makes a decision that the *outgoing* VM will be migrated, for example, in order to release its current *outgoing* computing node and eventually power it down.

2. Using the MON_if interface, the GSO fetches the parameters of the *outgoing* VM and, with the help of the GM_mgr interface, detects the UPA1 and UPA2 ranges that are linked to respective disaggregated memory sections. By looking up the current configuration of the interconnect, from the UPA1 and UPA2, the GSO derives the UPA6 and UPA7 ranges, that need to be attached to the incoming VM.

3. Based on the obtained parameters, the GSO selects an *incoming* computing node that is capable of attaching UPA6 and UPA7 as well as meeting other requirements of the *outgoing* VM.

4. The GSO triggers the *incoming* computing node to spawn a new VM process but in the *incoming mode*, that is, it will be initialized and waiting for the *outgoing* VM's state to be provided. At this point, it is important that the GSO obtains a handle (for example an IP address and port) at which the *outgoing* VM is waiting. Also, with the help of the GM_mgr of the *incoming* computing node, it discovers the UPA3 and UPA4 ranges which need to be linked to proper sections of disaggregated memory.

5. The GSO modifies the interconnect configuration in order to connect the UPA6 and UPA7 ranges to UPA4 and UPA3, respectively. For a short duration the UPA6 and UPA7 ranges are attached by two computing nodes but the *incoming* VM is not executing yet so this is harmless.

6. The GSO passes the address of the awaiting *incoming* VM to the *outgoing* one, using the `MON_if`. Upon reception acknowledgment, the GSO is sure that the *outgoing* VM stopped its execution and started sending the snapshot of its state to the *incoming* VM. The GSO can then safely tear down the connections of the `UPA1` — `UPA6` and `UPA2` — `UPA7` pairs. This is the last step performed by the GSO.

7. The transfer of the `VM_state` is performed directly between the *outgoing* and *incoming* VMs. Once this is done, the *outgoing* VM process terminates and the *incoming* one resumes the guest execution stopped before the migration.

Thanks to the fact that all guest memory comes from the disaggregated pool, only interconnect paths between a VM and respective memory buffers (including the shared ones) need to be updated during `VM_state` migration, instead of performing an actual `VM_payload` transfer. Only the *guest metadata*, which volume is typically quite small, are copied between *outgoing* and *incoming* computing nodes, so the operation should take very short time. Another important point is the minimal GSO involvement. Its mediation is indispensable at the operation setup phase but eventually, the copying itself is done between two computing nodes, independently from the central module of the system. Otherwise this could be a scalability bottleneck.
An alternative approach would be to migrate the `VM_state` between two computing nodes over a temporarily established shared memory. Again, this removes the necessity of having the nodes connected, but on the other hand, the GSO will have to perform additional steps (setting the shared memory before migration and tearing it down afterwards).

VM payload migration

As illustrated on figure 4-7b, `VM_payload` migration is defined as copying the part of a VM's memory footprint residing in a disaggregated memory pool. We assume that the transfer is performed from one *outgoing* memory node, selected to be unloaded, to one or multiple *incoming* ones, which will take over migrated data of the guest. The operation is performed by the *remote direct memory access* (RDMA) engine, a

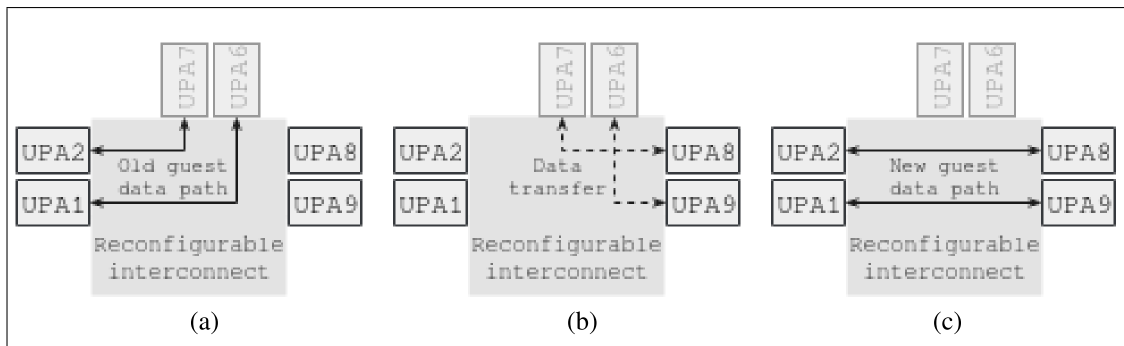


Figure 4-8: Interconnect reconfiguration steps

hardware component located at the *outgoing* memory node, optimized for rapid memory transfers between two memory nodes. The rationale here is that the main purpose of `VM_payload` migration is global data defragmentation, where a specific memory node (the *outgoing* one) is selected to be powered-down, provided that few remaining regions still in use could be migrated to other memory nodes. This way, it makes more sense to always program a RDMA engine of the *outgoing* node, instead of one or multiple engines of *incoming* nodes, which would be an alternative approach.

The `VM_payload` migration is conducted in the following steps (using the examples from figures 4-7 and 4-8):

1. The GSO makes a decision that the `UPA6` and `UPA7` ranges should be migrated to other memory nodes, for example as part of a periodic defragmentation task. At the beginning, the configuration of the interconnect paths looks like shown on figure 4-8a.

2. The GSO looks up its internal database to discover one or multiple VMs using these ranges as well as the `UPA1` and `UPA2` that they are connected with.

3. The GSO reserves ranges `UPA8` and `UPA9` at other memory nodes, capable of receiving the migrated data.

4. Through the `GSO_if` interface, the GSO stops the execution of a VM (one or multiple), which data are about to be migrated, and establishes proper connectivity between *outgoing* and *incoming* memory ranges, as shown on figure 4-8b. After that, the GSO programs the RDMA engine of an *outgoing* memory node and triggers the data transfer.

5. Once the data migration is done, the GSO modifies the interconnect for the second time, as shown on figure 4-8c, so that a VM is again capable of accessing its data.

6. Finally, the VM's execution is resumed.

In the presented situation, the amount of data to be transfered will be typically much bigger than in the `VM_state` migration case, for example a typical VM can use dozens of gigabytes or RAM, although not necessarily all of them would come from the same memory node. An important aspect is that the operation is performed directly between two memory nodes and no computing node is actually involved in it, except for a VM being temporarily stopped. Additionally, an RDMA engine handling the transfer can read the locally available data (at the *outgoing* memory node it is part of) and write them to UPA ranges corresponding to the *incoming* memory node. This does not require any intermediate data copying, which would otherwise have a significant performance cost. The presented approach is a simple one, in which the guest execution is stopped for the time of the operation (*stop-migrate-resume*) in order not to break guest data coherency. In this case, the *VM payload* migration boils down to properly managing the mapping of UPA ranges and transferring data but the virtualization layer remains unmodified. Enhanced approaches could potentially employ various *live VM migration* techniques (for example *live checkpointing*, *pre-copy* or *post-copy* migration [24, 25, 29, 41]), adapted for a disaggregated system, in order to reduce the downtime perceived by the guest.
With regard to memory sharing, if a migrated `VM_payload` contains a shared range, then all affected VMs have to be stopped during the migration until the interconnect is properly updated (this is subject to the same caveat as *live migration* techniques).

4.8 Chapter conclusion

In this chapter we expanded the design of flexible guest memory provisioning in order to provide the support for inter-VM memory sharing and VM migration. We proposed the interaction interface allowing to share a region of disaggregated memory between different VMs. Moreover, we discussed the subjects of memory or CPU migration and how it can support server consolidation efforts. In Chapter 5 we will change the focus

and move to the problem of configuring disaggregated devices in a direct attachment mode.

Chapter 5

Disaggregated peripherals attachment

5.1 Chapter introduction

In order to satisfy the requirements of workloads executed in a virtualized environment, in addition to memory provisioning that we broadly discussed in Chapter 3 and Chapter 4, another crucial aspect is related to the sharing of the various devices available across the system. In case of conventional server architectures, a very popular way of providing a device to a *Virtual Machine* (VM) is the passthrough method, which gives a VM an exclusive control over a device with no intermediate software emulation. This allows the guest device driver to access the device registers directly with near-native performance.

As mentioned in Section1.5 the architecture disaggregation has many advantages in general. It allows to increase the resource utilization factor, improves the resource balancing flexibility and potentially decreases the system operational costs. However, the passthrough technique cannot be implemented in its traditional form on a disaggregated system, as will be explained in Section . Therefore, it is highly desirable to adapt the technique so that the architectural switch to disaggregation does not imply an inferior performance of devices virtualization.

We propose a method and associated components necessary to re-enable the passthrough technique on a disaggregated architecture. Thanks to that, a guest device driver code, running on a computing node, is capable of accessing the device registers directly as well as the device can directly access the guest's RAM located on a remote memory

node. This constitutes the third main contribution of this dissertation.

5.2 Device emulation and direct attachment

A direct attachment, also known as device passthrough, is a device virtualization technique. In its principle, a dedicated *passthrough driver*, upon a VM request, maps the registers of a device (that is ranges of *Host Physical Address space* (HPA)), to the *Guest Physical Address space* (GPA). Thanks to that, the guest's device driver can read and write them directly, without any further host mediation, as illustrated on the right side of figure 5-1.

Otherwise (left side of the figure), a VM must emulate the device in software and serve as an intermediate layer between a *guest device driver* and a *host device driver*. This introduces additional performance overhead because it involves traps to the hypervisor, execution of emulation code and it may require data copying between guest and host.

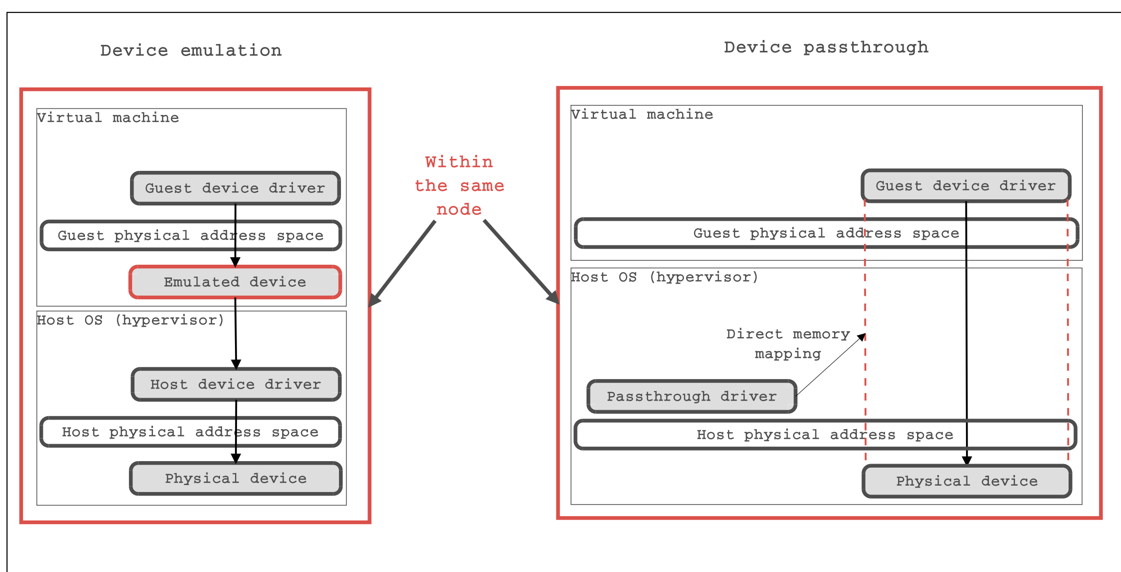


Figure 5-1: Direct attachment compared to device emulation

One downside of the device passthrough method is that a device can be attached to only one guest at a given time. Nevertheless, with hardware-provided device multiplexing, like the PCIe SR-IOV mentioned in Section2.5, the problem is well mitigated. An SR-IOV device can be efficiently shared between multiple guests, almost as if several separate devices were available.

Limitations in the context of disaggregation

The direct device attachment cannot be used for disaggregated peripherals in a straightforward way: switching to a disaggregated architecture changes fundamental assumptions underlying this method, as depicted on figure 5-2.

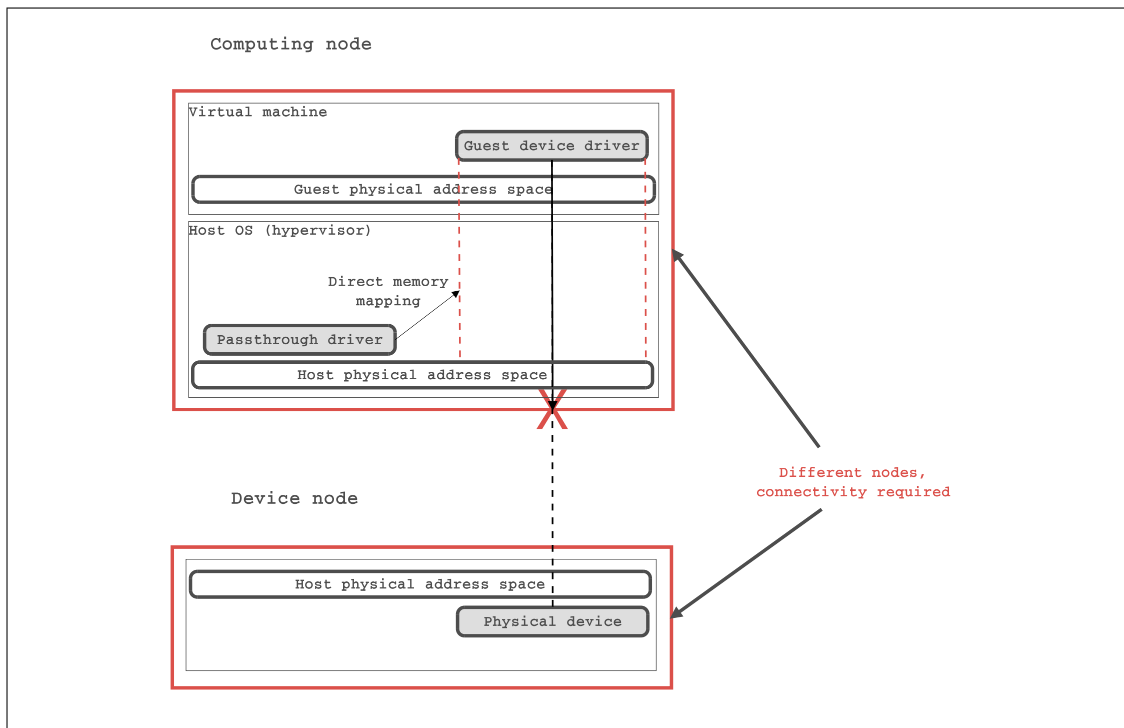


Figure 5-2: Standard passthrough not possible on disaggregated system

1. The device is physically attached to a remote device node and therefore its registers are mapped to a different HPA than the one visible by the *passthrough driver*.
2. A device may wish to perform direct accesses to the guest's memory, while it is located on yet another node (this is not shown on the figure for simplicity).

In order to workaround these limitations and enable direct device attachment on a disaggregated architecture, the driver must cooperate with other system components. Specifically it needs their help to setup proper connectivity between the involved system nodes and create proper address mappings.

Disaggregation challenges

The first requirement of a design proposal is the fact that disaggregation should stay transparent to a guest device driver. Indeed, ideally, the same driver should be used both

on disaggregated and clustered systems. It is also the same driver that would be used on the host with a device attached locally. Because of this requirement, all necessary modifications are performed by the VM, the passthrough driver and the *Global System Orchestrator* (GSO). The guest remains oblivious to that.
In order to specify further design requirements we first analyze the behavior of a generic (without specify any particular type of a device) guest device driver. Since it should remain unchanged, the analysis indicates functionalities expected from other system components.
Assuming that a device is successfully configured in the passthrough mode, the principle of operation of the driver is the following:

1. A user-space workload requests the driver to launch a device operation; the workload also indicates the type of operation and GVA pointers to input and output data buffers.

2. The driver translates the addresses of the buffers from GVA to GPA because only these are valid for a device's DMA engine.

3. Input data is written to the proper device registers, also mapped to the proper ranges of the GPA, and this schedules the device processing.

4. Once the processing completes, the driver receives an interrupt, at which point output data is available in the previously indicated output buffer.

This short overview already reveals the following challenges:

Ch.1 The device control registers are mapped to the HPA range of the device node, not to the HPA range of the computing node. Therefore, the latter must reserve a corresponding range locally and connect it with the former.

Ch.2 All the guest RAM comes from a memory node, which ranges are linked with respective ranges of the computing node, as explained in Chapter 3. The GPA addresses generated by the guest driver are mapped to HPA addresses of the computing node, which allows to reach the correct HPA addresses of the memory node. In order to allow the DMA engine to access the same guest RAM, there

must be similar ranges allocated at the device node and linked with the same destination HPA ranges of the memory node.

Ch.3 The peripheral is not the only component in the device node, there are three complementary components:

- an actual device processing unit (PU), which performs the computation,
- a DMA engine, used to fetch input data from guest's memory and to store output data to guest's memory — all this is done by hardware and does not require any action from the CPU,
- an *Input-output memory management unit* (IOMMU), processing all memory accesses performed by the DMA engine. Addresses emitted by the DMA engine are translated before reaching the memory buffers. The translation maps must be programmed beforehand by the device driver, otherwise the accesses are not allowed. Therefore, this unit serves both for mapping purpose and to prevent the device from accessing memory regions it is not authorized to access.

The first two pieces of hardware are commodity components and may be implemented as one single or two different units. The third one is specific to the disaggregated architecture and therefore presented as a separate system module. Since the DMA engine operates on GPA addresses generated by the guest driver, it is therefore the role of the IOMMU to translate these addresses to UPA addresses of the device node, which are linked with respective ranges on the memory node. Guest-specific address translation maps have to be properly build during the device attachment process.

Ch.4 The communication between a guest driver and the device is based on the former reading and writing device registers and the latter sending interrupts. An interrupt is usually received in the host by the *Interrupt Controller* (IC), a piece of hardware supporting an operating system in proper interrupt servicing. For example, if an interrupt controller supports virtualization, a hardware interrupt may be directly dispatched to the guest, with no return to hypervisor. Since a device is attached

to a different system node, it has no way to plug its interrupt line to the IC, as it is expected. Therefore, an additional interrupt routing is needed so that the IC of a computing node can receive interrupts originated at remote peripherals, as if they were attached locally.

The management of the interconnect paths, the selection of the corresponding *Uniform Physical Address space* (UPA) ranges and the establishment of the corresponding IOMMU translation maps are beyond the capability of the guest device driver or of a single VM. Thus, a disaggregated device attachment and detachment must be performed in cooperation with the GSO, as presented below.

5.3 Disaggregated passthrough design

The following part of this chapter is the core of a novel design enabling the passthrough attachment on a disaggregated system. The design is part of the patent application prepared by us and filed in November 2018. Therefore this is a confidential part of the dissertation and was moved to Appendix A

5.4 Chapter conclusion

In this chapter we explained why the passthrough method cannot be used on a disaggregated system in its default shape. Moreover, in Appendix A we proposed how to adapt the passthrough method in order to remove such limitation. That was the last chapter providing core contributions of this work. In the following, Chapter 6 evaluates the prototype of a design described in Chapter 3 and Chapter 7 concludes this dissertation.

Chapter 6

Implementation and evaluation

This chapter presents a prototype implementation of a system based on the architecture presented in the previous chapters. It provides a preliminary performance evaluation, in the sense that a real disaggregated system was not yet available at the time of gathering performance numbers. The best possible solution was to emulate such a system. Thus, its scale is not the same; there are only a few gigabytes of memory available, instead of terabytes. Additionally, it was not possible to observe potential throughput or latency overheads introduced by a real communication network between disaggregated system nodes. Despite its limitations, the presented prototype allows to assess the functional correctness and the technical feasibility of the proposed software architecture. It also allows to investigate the performance overheads related to the various software tasks of our architecture.

6.1 Memory provisioning

The prototype targets the memory provisioning and the flexible guest RAM resize introduced in chapter 3. Relevant system components are illustrated on figure 6-1.
The work was conducted within the *dReDBox* project, which target platform was ARMv8 [56]. Nevertheless, except for some parts of the memory *hot-plug* functionality of the guest kernel, the rest of the implementation is platform-agnostic. The prototype is based on open-source software: the *Linux* kernel running both as host and guest OS and *QEMU* being the host user-space process that communicates with the KVM Linux

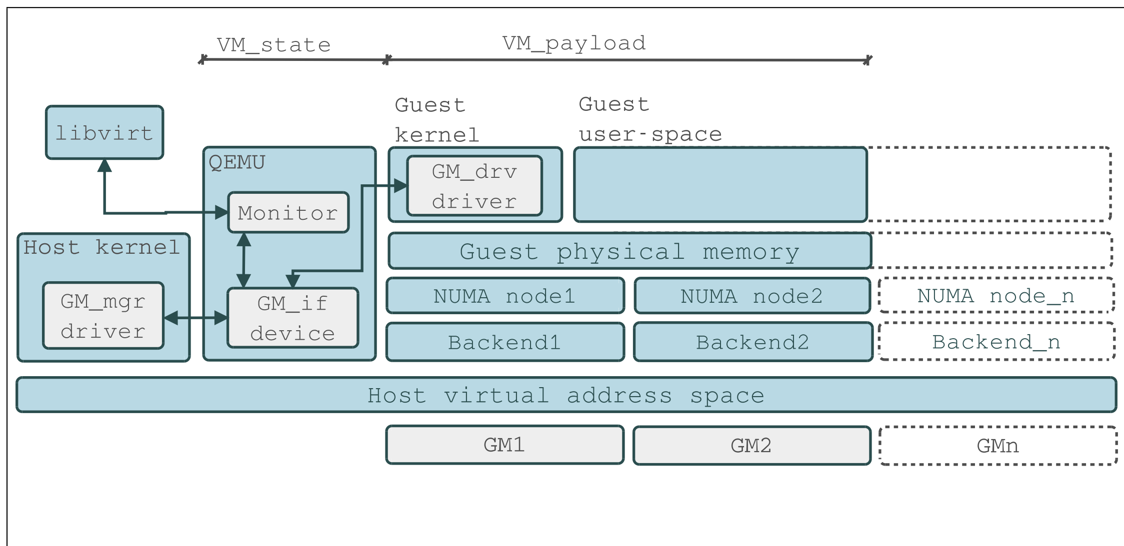


Figure 6-1: System prototype components

driver in the host and handles VM emulation.

The runtime adjustment of the guest RAM is enabled without relying on the ACPI standard [9] support. Indeed, some operating systems on some platforms do not support the ACPI standard at all [2], or the support may rely on a *closed-source* proprietary firmware. The latter was intentionally avoided in the context of the *dReDBox* project as well as also to guarantee a broader scope of the research outcomes.

Instead, in the presented implementation, the interaction between a guest and an underlying VM is handled in a paravirtualized manner. Paravirtualization is a technique especially popular for guest device drivers, where the driver is split in two parts. The *front-end* is installed in the guest and exposes an API expected by a client workload. However, all the processing is delegated to the *back-end*, installed in the host, that eventually performs the intended task, for example, operating the underlying hardware. An example implementation is the *VirtIO* [50] framework using a memory region shared between the host and the guest as a communication medium.

KVM is the *Linux* kernel module that enables virtualization features by reusing the infrastructure already present inside the kernel and turning it into a hypervisor. As a driver, KVM exposes only the minimum required functionalities. It needs a complementary user-space client providing a virtual machine emulation to form a full-featured virtualization framework capable of launching guest operating systems.

The user space component is *QEMU*. *QEMU* provides an access to various platform

hardware (for example disk storage, network interface card). It also initializes memory buffers which, from the guest's perspective, are perceived as physical RAM. As a matter of fact, *QEMU* is perhaps the most popular and feature-rich KVM client but it is not the only one. For example, *kvmtool* has a much smaller code base and is often used by KVM developers for testing purposes [13]. Another similar piece of software is *crosvm*, used to run Linux virtual machines on Chrome OS [11].
QEMU interacts with KVM through the `ioctl()` system calls and the information passing between the guest OS and QEMU is based on emulated virtual devices exposed to the guest as GPA ranges.

Implementation details

The *QEMU* process (`VM_state`) runs in the host local memory and only the resources allocated for a `VM_payload` are mapped to isolated memory chunks, as described in section 3.2. This mapping is performed by the `GM_mgr` host driver that, based on information from the *device-tree* structure, initializes subsequent `GMk` ($k \in [1, 2, \dots n]$) GPA ranges as a custom flavor of *reserved memory* (in *Linux* kernel parlance). These `GMk` GPA ranges are not passed to the system memory allocator (the *buddy allocator* in Linux). This effectively provides memory isolation since the access can be granted only by the `GM_mgr` driver. For the purpose of communication with user-space, the driver exposes the `GMk` ranges as separate character devices, each one of *guest memory section* size (512 MB for the prototype purpose). On *QEMU* side, the crucial modifications developed for the system prototype are the following:

1. A reused custom backend, based on existing ones created for *non-uniform memory access* (NUMA) abstraction. The backend allows to construct resizeable guest memory out of preallocated buffers instead of obtaining it from the host memory allocator (the default behavior). Instances of the backend (`Backend_`k) acquire resources from `GM_mgr`, which maps `GM_`k ranges to the virtual address space of a given VM process.

2. A new command in the *QEMU monitor* that allows to ask for guest memory resize. The monitor is a VM service exposing an interface (`MON_if`) typically

used for VM diagnostics and orchestration by external tools (`Monitor`, for example *libvirt* [5]).

3. A modification of the control register of a `virtio-balloon` emulated device, necessary for communication with the respective driver at guest side. This is a device implementing the *memory ballooning* technique on top of the *Virtio* framework and providing a communication mechanism based on unidirectional queues located in the memory shared between the host and the guest. The *memory ballooning* mechanism is not used but it was just convenient to reuse an existing virtual device and its driver for the paravirtualized communication.

4. A custom state-machine that takes care of guest-to-host (`GM_if` to `GM_drv`) event signaling and correct data transfer, also placed within the `virtio-balloon` implementation.

Additionally, on the guest kernel side, the `virtio-balloon` driver was properly modified (`GM_drv`), correspondingly to the device part described above.

Linux memory hot-plug patch

As a crucial component of the system, the missing parts of the memory *hot-plug* functionality in the *Linux* kernel were developed. A generic hot-add/-remove framework had been already present in the Linux kernel but the support for the ARMv8 platform-specific steps was missing. This part of the work has been published to the Linux community in the form of code patches [14]. Initially, we only implemented the *hot-add* part, then another version of the patch included also the *hot-remove* functionality and this one was released once again, with additional fixes included. Our contributions received some comments from the community as well as we few industry partners expressed their interest in the status of this feature. This proofs that our work has a good chance to be merged into upstream version of the kernel in the future. However, for now it is not and there are two main reasons for that. Firstly, the patch is a research work and it is still not perfect. The *hot-remove* crashes occasionally on some boards and it is not trivial to find the reason why. Secondly, the kernel maintainers do not seem to be strongly motivated to accept this patch because there are no well-known

contributors (or industry partners) behind it, who claim that they really need the feature and will maintain it in the future. The Linux kernel development is typically driven by demand from the industry in order to limit the growth rate of the (already huge) codebase volume.

Testbed configuration

The prototype implementation of the presented architecture has been tested on the *Xilinx Ultrascale+ MPSoC ZCU 102 (rev. 1.0)* board, equipped with 4 *Cortex-A53* ARM cores and hosting 4 GB of external memory [10]. The host OS was based on a manufacturer's fork of the *Linux* kernel (version *xilinx-v2016.4*), *QEMU* on upstream version *stable-2.5* and the guest *Linux* kernel on upstream version *v4.14-rc8*.
The *device-tree* used by the host OS was modified in order to split the available RAM in two parts: 2 GB backing up the *local memory* and 2 GB serving as *isolated backends*. The system was configured so that the size of the guest memory sections is 512 MB. For example, when a VM is started with only one section, it can be scaled up to 2 GB in 3 steps of one added section each.

Memory resize latency

	Stage	**Min [us]**	**Max [us]**	**Median [us]**
Add	backend reservation	140690	147315	145794
	page table build	89	62664	30254
	section init	32349	38923	32472
	pages onlining	64802	87691	80878
	guest subtotal	**97331**	**182258**	**142914**
	request-reply	113411	206678	167870
	total	**254541**	**353043**	**313403**
Remove	pages offlining	57173	720117	127242
	section clean-up	1868	10462	2489
	page table destroy	30049	33135	30206
	guest subtotal	**89551**	**756366**	**160055**
	request-reply	101252	806490	197394
	backend release	122196	127283	126329
	total	**224368**	**933459**	**323071**

Table 6.1: Latencies of memory resize steps

In order to investigate the resize request path, a VM was started and orchestrated through the *QEMU monitor* by a simple script. Initially launched with 512 MB of RAM, the VM was scaled up to 2 GB in 3 steps (512 MB each), then scaled down in 3 other steps to return to the initial configuration. After each step the script was waiting a few seconds before triggering the next resize. There were 40 rounds conducted of 6 such steps each. Multiple time samples were collected from both *QEMU* and the guest kernel side. Subsequently the samples were processed to obtain the statistics presented in table 6.1. The table shows the delays contributed by each resize stage. It is split in two parts: the *scale-up* (upper half) and the *scale-down* (lower half).
The resize request is always received first by *QEMU*, which triggers further steps and receives an acknowledgment at the end (as depicted on figure 3-2). The *total* latency is derived from these measurements. It consists of the *backend reservation* or the *backend release* at host side, plus the *request-reply* latency perceived by *QEMU*, that is the communication delay and the respective reconfiguration delay at guest OS side (*guest subtotal*). The communication is based on accessing the device registers together with handling the related interrupt and can be simply computed as:

$$communication_delay = request_reply - guest_subtotal$$

The order of execution is reflected within both *Add* and *Remove* sections of table 6.1. For the *scale-up*, memory *backends* are reserved before the guest can use them. Symmetrically, for *scale-down*, *QEMU* has to request the guest OS to relinquish the memory sections first, before the corresponding *backends* can be safely released. The signaling overhead is larger in the *scale-down* case because of the additional interaction that is necessary at the beginning to trigger the guest action first.
The ***guest subtotal*** values can be decomposed in three main steps, done symmetrically for the *Add* and *Remove* part:

- Building or thrashing the page tables used by the *Memory Management Unit* (MMU),
- Initialization and cleaning of the virtual memory map (so called *vmemmap*) data structures,

- Pages onlining or offlining, that stands for passing or withdrawing pages to/from the system allocator and flushing the *Translation Lookaside Buffer* (TLB) entries in the offlining case,

Several stages with a particularly high latency variance deserve more comments.
First, the page table building requires to fill the entries of the 4-fold-nested address translation tables (this is the default configuration in the *Linux* kernel for ARMv8), which *tree-like* structure is determined by the MMU. Each entry of a level-*N* table (that is one memory page) corresponds to one table of level-*(N+1)*. The level-1 table and some lower level tables are allocated at boot time, but in order to increase the amount of addressable memory, further pages may need to be allocated for lower levels. The deeper the level is, the faster the corresponding table is populated and the more frequent additional allocations are. Therefore, three subsequent allocations for one mapping may account for the worst case latency and none of them for the best case.
Another large variance is observed in the case of the *pages offlining* stage, where pages of a given section are reclaimed. The pages to be reclaimed may be owned by the guest memory allocator or entailed on *per-CPU* lists of each CPU core (in *Linux* a core is considered a separate CPU). The operation may not be successful at the first attempt, especially in the second case. The operation is thus repeated on each core until it is accomplished, otherwise the process cannot advance. Depending on how long the system was running and on the allocations and deallocations performed by the different cores, the number of pages of a given section owned by per-CPU lists ranges from 0 to many and the introduced delay can also vary.
The *section init* and *clean-up* stages embody the virtual memory map (so called *vmemmap*) management, that is instantiation and clean-up of `struct page` objects, respectively. Although the delay introduced by the *clean-up* step is also variable in the removal case, it does not significantly contribute to the overall latency.
In general, the execution time when adding the memory section is typically above 300ms, up to above 350ms in the worst case. Out of this, around 45–50% is spent in the guest, 40–45% in *QEMU* and about 8% is consumed by the communication between them. The biggest variability is related to the page table building stage. The median execution time of section removal is 320ms, with *QEMU* accounting for about 40% of

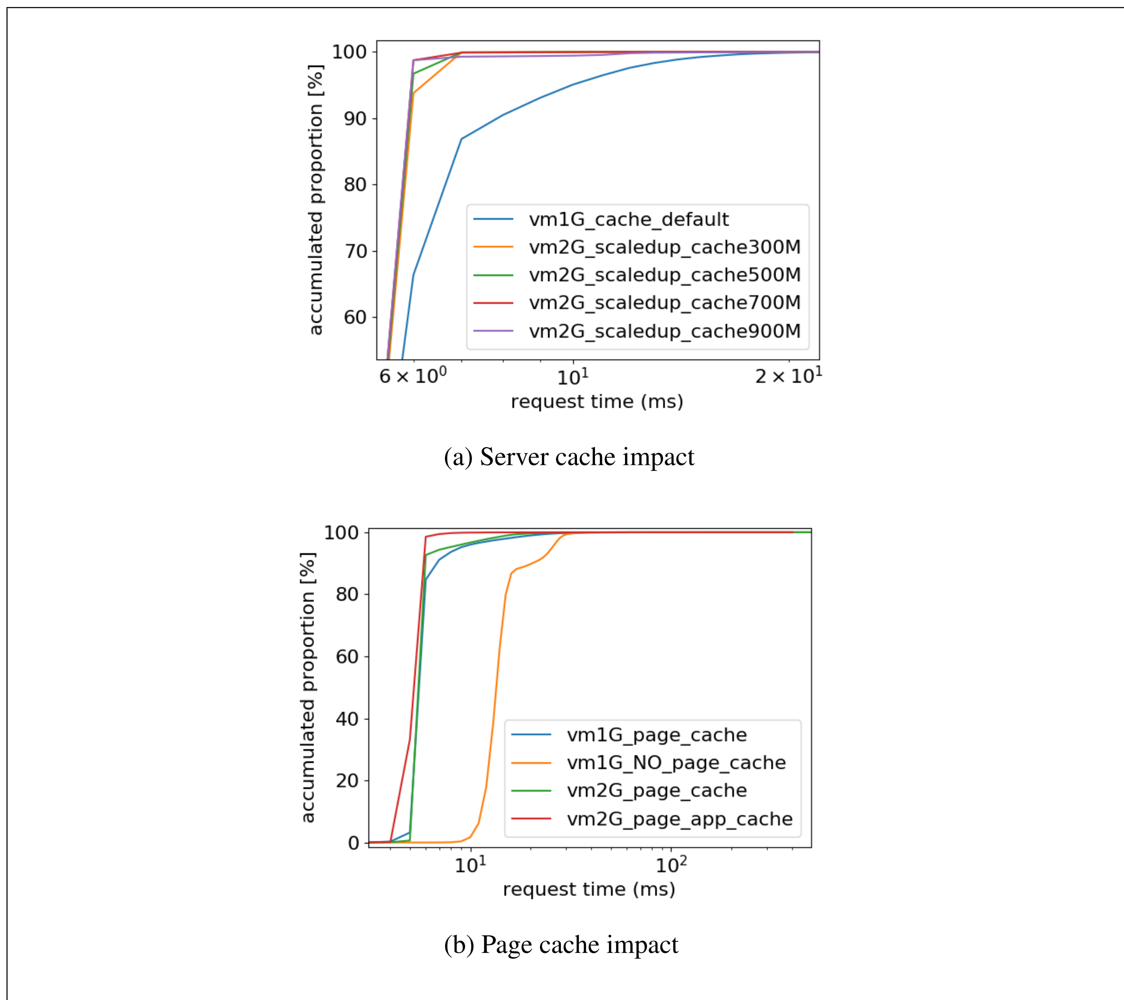
it, guest for 50%, and more than 11% devoted to communication. But the dispersion of this operation is much larger at the guest side: in the worst case it can take almost 5 times longer than the median value, mainly because of the page offlining latency. The numbers in table 6.1 are derived statistically from the set of samples for each stage so the above percentage values do not add up exactly to 100%.

It should be noted that these numbers do not consider the situation where some *outgoing* pages of a section being *hot-removed* are still in use. In such cases, free pages from other sections have to be found to substitute them and data must be copied. This can further delay the *offlining* stage or, if substitution cannot be found, the whole operation can be refused. This is a good example of how memory fragmentation can be harmful and why it should be prevented.

Eventually, a crucial aspect is how much a guest workload would be affected by *scale-up* and *scale-down* operations. At guest side one CPU core is involved in the whole reconfiguration process. Other cores can still execute the workload in parallel. The only exception is the *offlining* step, where all cores have to release their respective pages from their *per-CPU* lists. Similarly, the workload does not have to be stopped while QEMU is reserving or releasing *backends*. Therefore, the downtime perceived by workloads can be partially mitigated, provided that it is capable to actually advance its execution without additional memory resources.

SimpleDB use-case

As an example use-case of dynamic memory resizing we measured the performance of a custom in-memory key-value database storing pairs of *[domain name, TLS certificate]*: the *SimpleDB* example application. *SimpleDB* is compatible with the *Redis* API. This allows to evaluate its performance with a standard `Redis-benchmark` program. It is also possible to resize its internal cache using the `Redis-CLI` tool [16]. Our benchmark issues 100K `GET` requests in total. Each request queries the certificate of a key containing a randomly chosen 12-digit value in the `[0, 200K[` range. To answer a request the database primarily looks up its internal cache (of tunable size). If that fails, it reaches out to disk to obtain the value from a file. Note that in order to reduce the amount of I/O operations, the *Linux* kernel leverages on the aforementioned *page*



(a) Server cache impact

(b) Page cache impact

Figure 6-2: Latency results from `Redis-benchmark` against the *SimpleDB* application

cache. Therefore, there are 2 types of software caching: *server-internal* and the *page cache*.

Figure 6-2a presents the request service latencies that were measured during the experiments. For better readability only the part of the results exposing performance differences is presented on the plot.

The configuration for the first benchmark presented on figure 6-2a (`vm1G_cache_default`) is the following. At start, the server has been launched inside a VM with 1 GB of RAM with the internal cache set to 100 MB. The mock database had been filled with 200K generated certificates (each of size 4096 B). The *page cache* usage (observable with the `free` command for example) was then occupying 280 MB. Subsequently, the guest's RAM has been extended up to 2 GB by *hot-adding* two memory sections. Then, the server's cache has been gradually increased in several steps (200 MB each). After each

step the `redis-benchmark` has been launched. During this operation the *page cache* occupancy did not change, which means that the server was able to increase its internal cache by exploiting the memory attached at run-time and therefore still benefit from the cached I/O operation results when necessary. The mean response time improved. For example, the first configuration was able to serve 87% of requests in up to 7ms, compared to 99% or more with bigger internal caches. Since the database was able to store more data internally, it spared time required for, at least, context switches (if data would be in *page cache*) or possibly disk I/O operations.

To better excercise the *page cache* impact, se performed another set of tests which results are shown on figure 6-2b. As in the first experiment the server has initially been launched in a VM equipped with 1 GB of RAM (`vm1G_page_cache`), but then an auxiliary application has been used to allocate most (600 MB) of the memory previously used for file caching (`vm1G_NO_page_cache`). For 90% of requests this resulted in the mean response time growing from 7ms to 22ms, which is more than 3 times slower. Then, the guest RAM has been expanded to 2 GB and another run (`vm2G_page_cache`) showed that the performance was very similar to the first setup (slightly better perhaps because the auxiliary application did not destroy the initial *page cache* completely and a small fraction did not have to be recovered again). On top of that, for the last test, the server cache was enlarged from the default 100 MB to about 780 MB and the mean response rate improved even more.

Chapter 7

Conclusion

This dissertation started with a presentation of system disaggregation and the rationales behind it, especially relevant for the data-center sector. In this context the main part of the work focuses on operating systems and virtualization enhancements. The proposed solutions are related to provisioning of disaggregated memory and peripherals to guests executed in VMs.

In Chapter 3, we described the design of a guest memory provisioning scheme enabling flexible RAM resizing at runtime. The solution makes use of dedicated HPA ranges isolated from the default memory allocator. Although the ranges are supposed to represent resources from remote memory nodes, the approach can be adapted on traditional systems as well. In fact, it was the only possible way of evaluating the prototype implementation, as presented in Chapter 6. The design allows to perform runtime memory balancing on a section granularity, which simplifies management logic and reduces memory fragmentation. The necessary communication between guest and host components is realized in a paravirtualized manner and does not require any specific software parts (for example it does not require the ACPI standard emulation). In this way, it is not bound to any specific virtualization framework or the hardware platform type.

On top of the memory provisioning scheme, Chapter 4 presented the concept of inter-VM memory sharing adapted to a disaggregated system. By design, regions of memory can be shared across the system in a zero-copy manner, regardless of an actual location of involved VMs. The approach relies on a disaggregated system infrastructure and

the *Global System Orchestrator* (GSO) to attach the memory to the host. On a VM level, several ways to attach a shared region have been discussed together with the serialization of concurrent accesses. Additionally, Chapter 4 explained why the VM migration term gained a twofold meaning in a disaggregation context and how this can open new opportunities for server consolidation.

Additionally to memory provisioning, Chapter 5 focused on devices disaggregation. It showed why disaggregated devices cannot be directly attached to VMs as it is possible on traditional architectures. Subsequently, we proposed an approach to compensate this limitation and to configure devices in the passthrough mode with the help of the GSO and the reconfigurable interconnect. Thanks to our scheme, a VM can operate the device as if it was attached locally to the host as well as the device itself can access regions of disaggregated guest memory using the GPA addresses. On top of that, the attachment configuration can be updated so that the design is fully compatible with the flexible guest memory provisioning presented in Chapter 3. Thanks to that, adopting the architecture disaggregation does not forbid the passthrough-attachment, which would otherwise be a great limitation.

Finally, Chapter 6 provided the evaluation of the resizeable guest memory prototype implementation. It proved that the memory resizing framework adoption is feasible, with respect to the operation overhead. The chapter also presented one use-case, representing a group of workloads taking advantage of the in-memory storage. Thanks to a flexible memory provisioning these workloads do not have to use secondary storage (disk) or be migrated to a VM having more RAM. Instead, they can benefit from a dynamic guest memory adjustment and continue execution without interruption.

Modifications performed at the hardware architecture level require corresponding adaptation of the software stack, including the virtualization support. Authors believe that what makes this dissertation useful and interesting (also outside of the scientific community) is the way how the subject of virtualization layer adaptation was explored.

Presented concepts are discussed on a theoretical level but they are also strongly related

to pieces of software widely used in industry (*Linux* kernel, *QEMU*). Compatibility with industry-level solutions allowed to provide a design that is feasible to deploy. We did our best to provide rationale behind various design choices as well as discuss alternatives. We managed to present a prototype implementation of flexible guest memory provisioning and expand the design further to provide the support for memory sharing and migration. With regards to devices provisioning we explained how a disaggregated system can be designed in order not to compromise a popular technique, which is the direct device attachment. Our concept requires both hardware and software components but, except for that, we did not enforce any modifications at the level of guest operating system or used devices.

Definitely, the weakest point of this work is the prototype evaluation part. Although we hoped to do it, for the lack of a disaggregated hardware prototype at the time of writing, we did not integrate the inter-VM memory sharing and VM migration features. Thus, we were not able to get any performance numbers that would allow us to asses the solution. Also the design of direct attachment of disaggregated devices was not implemented however it was not meant to be for the time constraints.

7.1 Perspectives and future works

As presented in Chapter 1, disaggregation is definitely an interesting approach in designing large scale systems. Together with solutions presented in this dissertation, its wide adoption by providers will be possible if it lowers the TCO so that the cost of investment is paid back in a reasonable time and generate a higher income than current systems afterwards.

Except for that, the adoption depends also on the availability of alternative solutions for challenges currently faced by providers. For example, there already exist *Non-volatile memory* (NVM) devices that can offer latencies only less than an order of magnitude higher than *Dynamic random-access memory* (DRAM) [23]. Assuming that its performance may improve while the technology matures, NVM devices may be used as a swap space, of an access latency few orders of magnitude smaller than a disk. Comparing to disaggregation, installing a new type of memory devices does not

affect the whole system architecture so heavily, which can be a very important decision factor considered by providers. Other relevant factors are energy consumption of NVM devices, their lifetime duration and finally the total investment cost.

As a future work, ideas presented in this work could be expanded in several ways.

Firstly, the section-based guest memory balancing scheme could be combined with the page-based one. However, as discussed in Section 2.1 the range from which pages would be selected should be limited in order not to increase the memory fragmentation. Secondly, it could be better analyzed how to reserve resources that can be used to add a section of memory when the system is already in the *out-of-memory* state. Switching from the *out-of-memory* state in this way spare the necessity to swap pages to disk or kill user space processes.

Further, the VM payload migration introduced in Section 4.3 could be integrated with *live migration* techniques. The *live migration* techniques allow a VM to continue execution while the data is being moved so that the operation is almost unnoticed from the workload perspective (partial performance degradation for a short period of time is typically accepted).

Finally, the whole design could be examined and enhanced from the security point of view.

For example the inter-VM memory sharing mechanism, introduced in Section 4.2, could be improved with the help of asymmetric cryptography. It is crucial that only authorized VMs are able to join a shared region. Thus, instead of a shared region identifier, VMs would obtain its ciphered version. In order to compute the deciphered identifier, a VM would need a key, which could be handed only after a successful authorization.

Bibliography

[1] Htc dc 3 whitepaper - online version, 2014. https://goo.gl/6pks7z.

[2] Acpi on arm64: Challenges ahead, 2015. https://events.static.linuxfound.org/sites/events/files/slides/acpi_on_arm64_0.pdf.

[3] "cloud service and deployment models", 2017. https://cloudcomputing.ieee.org/images/files/education/studygroup/Cloud_Service_and_Deployment_Models.pdf.

[4] High performance conjugate gradients, complementary to hpl, 2017. http://www.hpcg-benchmark.org/.

[5] Libvirt, the virtualization api library, 2017. https://libvirt.org.

[6] The linpack benchmark, used to rank top500 list, 2017. https://www.top500.org/project/linpack/.

[7] Nested paging - wikipedia source, october 2017, 2017. https://en.wikipedia.org/wiki/Second_Level_Address_Translation.

[8] Top500 supercomputers rank, november 2017, 2017. https://www.top500.org/lists/2017/11/.

[9] Advanced configuration and power interface - webpage, 2018. http://www.acpi.info/.

[10] Board data-sheet, 2018. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[11] crosvm - a virtual machine monitor for chrome os using kvm, 2018. https://chromium.googlesource.com/chromiumos/platform/crosvm/.

[12] Ivshmem spec, 2018. https://raw.githubusercontent.com/qemu/qemu/master/docs/specs/ivshmem-spec.txt.

[13] kvmtool - an alternative virtual machine monitor using kvm, 2018. https://elinux.org/images/4/44/Przywara.pdf.

[14] Linux kernel patch, "memory hotplug support for arm64 platform", 2018. https://lkml.org/lkml/2017/11/23/182.

[15] Rcu mechanism, wikipedia definition, 2018. https://en.wikipedia.org/wiki/HyperTransport.

[16] Redis, in-memory database, webpage, 2018. https://redis.io/.

[17] Uefi, unified extensible firmware interface, 2018. https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface.

[18] Ibm summit specification on top500, June 2018. https://www.top500.org/system/179397.

[19] Openmpi - project webpage, June 2018. https://www.open-mpi.org/.

[20] L Adam. Manage resources on overcommitted kvm hosts, 2011.

[21] Nadav Amit, Dan Tsafrir, and Assaf Schuster. Vswapper: A memory swapper for virtualized environments. *ACM SIGPLAN Notices*, 49(4):349–366, 2014.

[22] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[23] Amro Awad, Simon Hammond, Clay Hughes, Arun Rodrigues, Scott Hemmert, and Robert Hoekstra. Performance analysis for using non-volatile memory dimms: opportunities and challenges. In *Proceedings of the International Symposium on Memory Systems*, pages 411–420. ACM, 2017.

[24] K Chanchio, C Leangsuksun, H Ong, V Ratanasamoot, and A Shafi. An efficient virtual machine checkpointing mechanism for hypervisor-based hpc systems. In *High Availability and Performance Computing Workshop*. Citeseer, 2008.

[25] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[26] Qualcom data-centers plans. `https://www.electronicdesign.com/industrial-automation/future-uncertain-qualcomm-loses-data-center-president`.

[27] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[28] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 10. ACM, 2013.

[29] Michael R Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 51–60. ACM, 2009.

[30] Rui Hou, Tao Jiang, Liuhang Zhang, Pengfei Qi, Jianbo Dong, Haibin Wang, Xiongli Gu, and Shujie Zhang. Cost effective data center servers. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 179–187. IEEE, 2013.

[31] Wei Huang, Matthew J Koop, Qi Gao, and Dhabaleswar K Panda. Virtual machine aware communication libraries for high performance computing. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 9. ACM, 2007.

[32] Woomin Hwang, Ki-Woong Park, and Kyu Ho Park. Reference pattern-aware instant memory balancing for consolidated virtual machineson manycores. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):2036–2050, 2015.

[33] Pavle Ivanovic and Harald Richter. Performance analysis of ivshmem for high-performance computing in virtual machines. In *Journal of Physics: Conference Series*, volume 960, page 012015. IOP Publishing, 2018.

[34] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014.

[35] Kostas Katrinis, Georgios Zervas, Dionisios N. Pnevmatikatos, Dimitris Syrivelis, Theonitsa Alexoudi, Dimitris Theodoropoulos, Daniel Raho, Christian Pinto, Felix Espina, Sergio Lopez-Buedo, et al. On interconnecting and orchestrating components in disaggregated data centers: The dredbox project vision. In *Networks and Communications (EuCNC), 2016 European Conference on*, pages 235–239. IEEE, 2016.

[36] Gaurav Kukreja and Supriti Singh. Virtio based transcendent memory. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 1, pages 723–727. IEEE, 2010.

[37] Mrugani Kurtadikar, Apurva Patil, Pooja Toshniwal, and Jibi Abraham. An inter-vm communication model supporting live migration. In *Cloud & Ubiquitous Computing & Emerging Technologies (CUBE), 2013 International Conference on*, pages 63–68. IEEE, 2013.

[38] Yaqiong Li and Yongbing Huang. Tmemcanal: A vm-oblivious dynamic memory optimization scheme for virtual machines in cloud computing. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 179–186. IEEE, 2010.

[39] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and

sharing in blade servers. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 267–278. ACM, 2009.

[40] Haikun Liu, Hai Jin, Xiaofei Liao, Wei Deng, Bingsheng He, and Cheng-zhong Xu. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1350–1363, 2015.

[41] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, and Xiaofei Liao. Performance and energy modeling for live migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 171–182. ACM, 2011.

[42] Hamid Reza Mohebbi, Omid Kashefi, and Mohsen Sharifi. Zivm: A zero-copy inter-vm communication mechanism for cloud computing. *Computer and Information Science*, 4(6):18, 2011.

[43] Héctor Montaner, Federico Silla, and José Duato. A practical way to extend shared memory support beyond a motherboard at low cost. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 155–166. ACM, 2010.

[44] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. The hipster approach for improving cloud system efficiency. *ACM Transactions on Computer Systems (TOCS)*, 35(3):8, 2017.

[45] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference*, page 16. ACM, 2018.

[46] Presentation on rack disaggregation with PCIe networking. `http://opencomputejapan.org/wp-content/uploads/2013/09/20130528_03_OCP-APAC_PCIe_SDN_ITRI.pdf`.

[47] Tapasya Patki. The case for hardware overprovisioned supercomputers. pages 36–38, 2015.

[48] MR-IOV extensions Presentation on PCI Express SR-IOV. *http://www.cs.nthu.edu.tw/~ychung/slides/Virtualization/SR-MR-IOV.pptx*.

[49] SMARTER2020 report. *https://www.telenor.com/wp-content/uploads/2014/04/SMARTer-2020-The-Role-of-ICT-in-Driving-a-Sustainable-Future-Decem _2.pdf*.

[50] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[51] Ahmad Samih, Ren Wang, Christian Maciocco, Tsung-Yuan Charlie Tai, and Yan Solihin. A collaborative memory system for high-performance and cost-effective clustered architectures. In *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, pages 4–12. ACM, 2011.

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

[53] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 69–87, 2018.

[54] Jun Suzuki, Yoichi Hidaka, Junichi Higuchi, Teruyuki Baba, Nobuharu Kami, and Takashi Yoshikawa. Multi-root share of single-root i/o virtualization (sr-iov) compliant pci express device. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 25–31. IEEE, 2010.

[55] Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Hecatonchire: Towards multi-host virtual machines by server disaggregation. In *European Conference on Parallel Processing*, pages 519–529. Springer, 2014.

[56] Dimitris Syrivelis, Andrea Reale, Kostas Katrinis, Ilias Syrigos, Maciej Bielski, Dimitris Theodoropoulos, Dionisios N. Pnevmatikatos, and Georgios Zervas. A software-defined architecture and prototype for disaggregated memory rack scale systems. In *Networks and Communications (EuCNC), 2016 European Conference on*, pages 235–239. IEEE, 2016.

[57] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. Marlin: a memory-based rack area network. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 125–136. ACM, 2014.

[58] John Velegrakis. Operating system mechanisms for remote resource utilization in arm microservers , may 2015. *Technical Report FORTH-ICS / TR-452*, 2015.

[59] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[60] Josh Whitney and Pierre Delforge. Data center efficiency assessment. *Issue paper on NRDC (The Natural Resource Defense Council)*, 2014.

[61] Sendren Sheng-Dong Xu, Chia-Hong Wang, Teng-Chang Chang, and Shun-Feng Su. Multi-root i/o virtualization based redundant systems. In *Soft Computing and Intelligent Systems (SCIS), 2014 Joint 7th International Conference on and Advanced Intelligent Systems (ISIS), 15th International Symposium on*, pages 1302–1305. IEEE, 2014.

[62] Saneyasu Yamaguchi and Eita Fujishima. Optimized vm memory allocation based on monitored cache hit ratio. In *Proceedings of the 4th Workshop on Distributed Cloud Computing*, page 8. ACM, 2016.

[63] Jie Zhang, Xiaoyi Lu, Sourav Chakraborty, and Dhabaleswar K DK Panda. Slurm-v: Extending slurm for building efficient hpc cloud with sr-iov and ivshmem. In *European Conference on Parallel Processing*, pages 349–362. Springer, 2016.

[64] Ke Zhang, Yisong Chang, Lixin Zhang, Mingyu Chen, Lei Yu, and Zhiwei Xu. saxi: A high-efficient hardware inter-node link in arm server for remote memory access. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 560–569. IEEE, 2016.

[65] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A Mckee, Paul M Carpenter, Petar Radojković, and Eduard Ayguadé. Main memory in hpc: Do we need more or could we live with less? *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(1):3, 2017.

Appendices

Appendix A

Disaggregated peripherals attachment - confidential part

This is a confidential part of Chapter 5. For the best presentation coherence it should be moved to Section 5.3.

A.1 Disaggregated passthrough design

Design components

Figure A-1 presents a detailed architecture of the system capable of attaching disaggregated peripherals in the passthrough mode.

The global addressing scheme is illustrated by the *UPA axis* on which all UPA ranges can be cast as non-overlapping sections. An interconnect path is established by the *Global System Orchestrator* (GSO) and expressed as a link between several pairs, which means that an access to one side is directly forwarded to the other. For example, for the purpose of device configuration, we assume that a guest has its memory already attached, thus the connection between `upa_c_ram` and `upa_m_ram` is represented as a solid line. Three newly established links are drawn using dashed lines. Finally, the dotted connections represent the *control network*, that is a network between the GSO and all system nodes, used to exchange configuration messages.

Internally, each system node is designed over its own *Host Physical Address space* (HPA). However, because this addressing is ambiguous in the global perspective, each range that can be accessed by other nodes is allotted with a unique UPA range. The translation is performed by the piece of hardware called *interconnect interface*, separate for each node, and respective translation maps are also programmed by the GSO, using the *control network*.

On figure A-1 address ranges consist of three parts separated by underscore characters:

```
<ADDRESS-SPACE>_<NODE-TYPE>_<ROLE-NAME>
```

The first part specifies in which address space the range is allocated and the vertical arrows between blocks express the mapping of a given range across multiple address spaces. The second part specifies the type of system node the range refers to: “`c`” stands for *computing* node, “`a`” for device node (the letter itself comes from the word *accelerator*) and “`m`” for *memory* node. The last part of the name refers to a particular

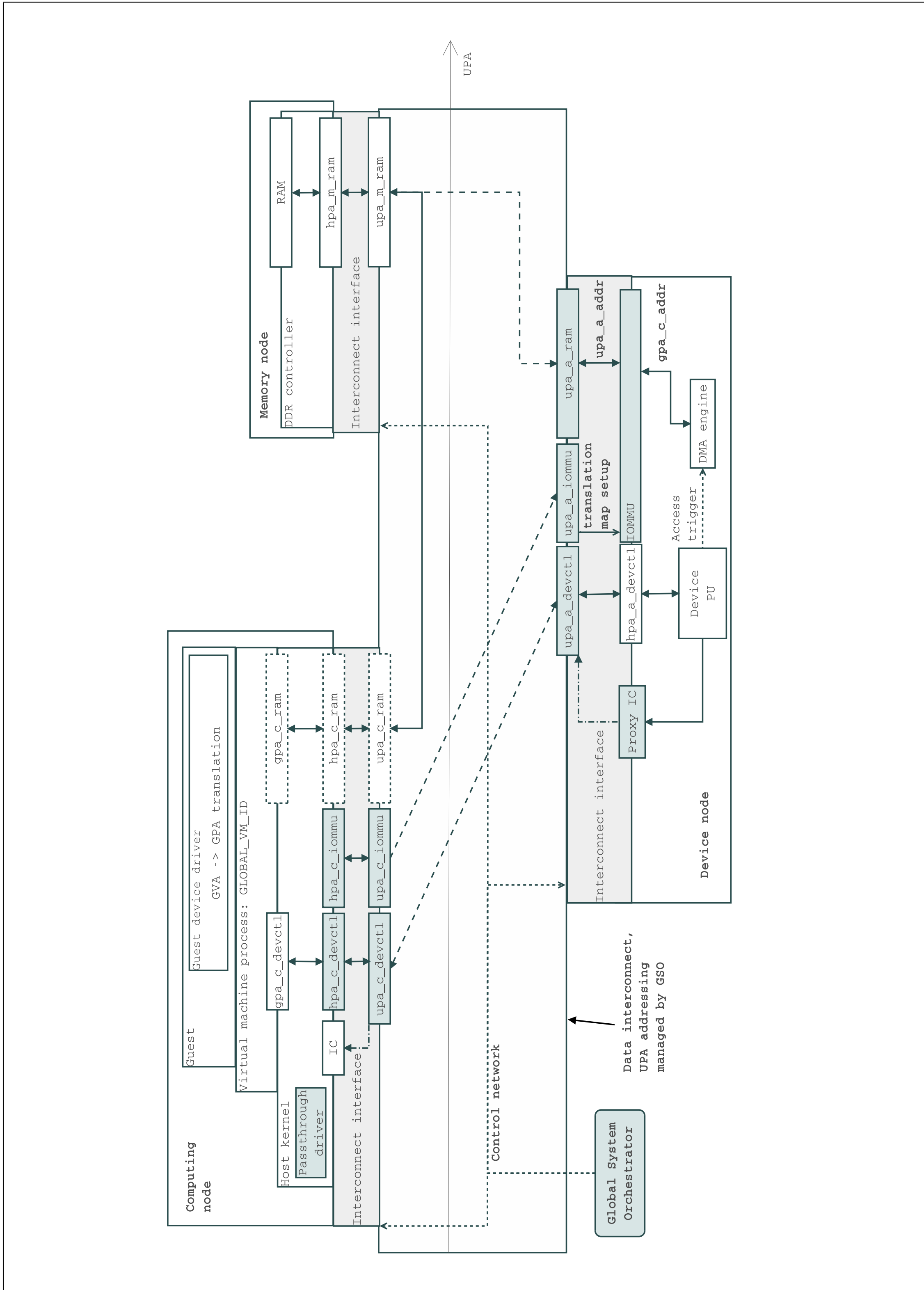


Figure A-1: Disaggregated passthrough — architecture design

role of the range and serves for the purpose of easier description. For example the upa_a_ram is an address range in UPA belonging to a device node and representing guest RAM. The role of other ranges will be explained in the next section, together with the attachment configuration. The lines between the different ranges with same role represent the interconnect path between them.

Some ranges can be established statically, for example upa_m_ram can be assigned at system start, based on the provided resources description. Additionally, provided that the system supports dynamic hardware discovery, it may also be assigned when a new memory module is plugged in. The situation is slightly different in the case of upa_a_ram as its size depends on the amount of guest RAM that a device should be able to access. On one hand, the maximum size and the number of such ranges could be upper bounded by the implementation. Nevertheless, for the design presentation purpose, we assume that the upa_a_ram is allotted dynamically. This is because the presented design also takes into consideration the possibility of a runtime RAM resize.

A *guest device driver* is a device-specific software module, which role is obvious. For a direct device attachment a crucial component is the *passthrough driver* running in the host system. During the attachment configuration, this driver is responsible for mapping the device registers, available as HPA ranges, to GPA. It has a full visibility of all hpa_c_ram ranges used by a VM. Therefore, it is capable of deriving the corresponding upa_c_ram ranges that will be passed to the GSO as a part of the attachment request, performed on behalf of the guest. The *passthrough driver* is common to all types of devices.

Infrastructure setup overview

In order to initiate a device attachment, there are several configuration steps to be done by different system components, as mentioned in Section . Referring to figureA-2, this section covers the GSO part.

At the beginning, the process starts upon request by the passthrough driver, a generic driver responsible for direct attachment configuration and agnostic to a device type. The driver communicates with the GSO on behalf of the VM and provides all necessary parameters (device type and others, skipped for a moment for better readability). Sub-

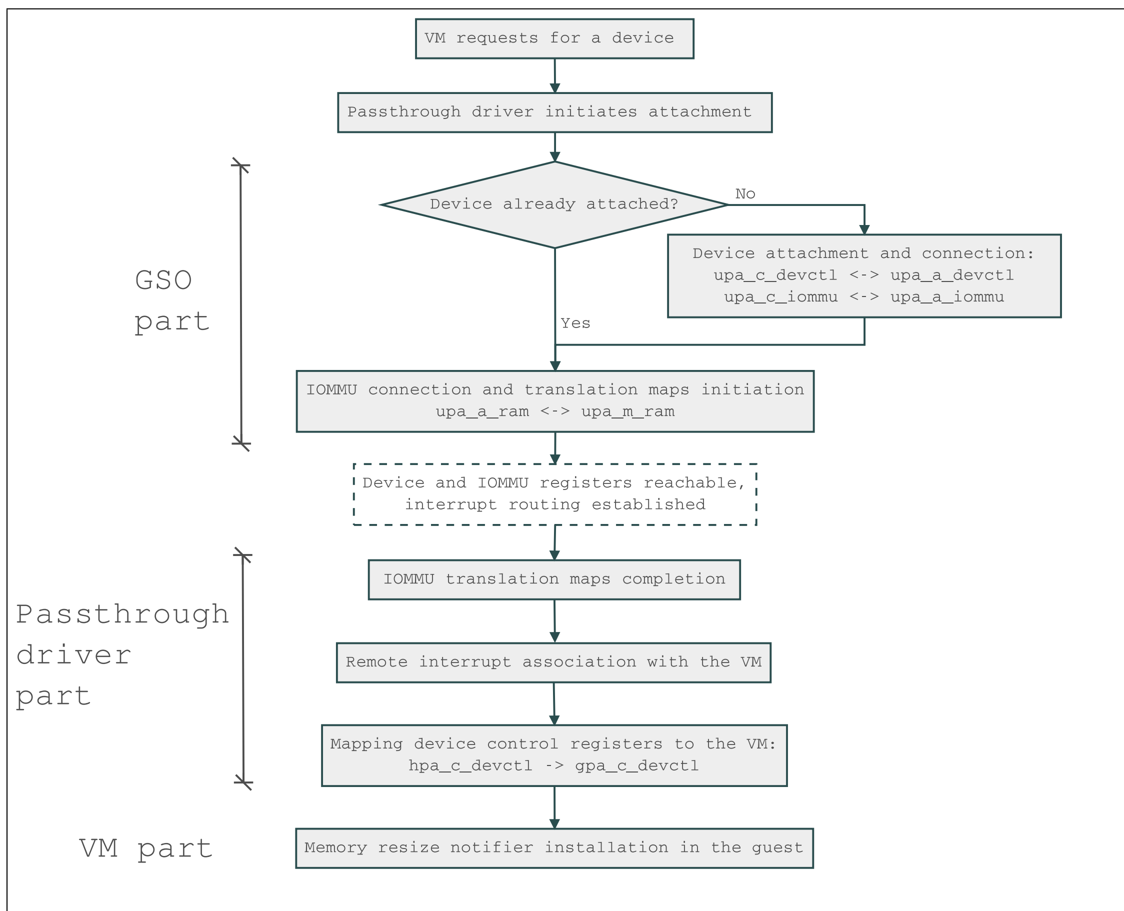


Figure A-2: Direct attachment initialization

sequently, the four main stages of a direct attachment configuration performed by the GSO are:

- Selection of a particular instance of the peripheral, according to the indicated type.
- Reservation of the UPA ranges for the computing node to attach the device registers, and for the device node to attach ranges of guest's disaggregated RAM.
- Initialization of the device's IOMMU translation map building so that addresses generated by the guest driver allow the device to reach the guest RAM on a memory node.
- Setup of the interconnect paths between proper upa ranges to enable data transfers between nodes.

When the attachment is completed, further configuration steps need to be done by the passthrough driver.

Passthrough driver setup overview

This section refers to the *passthrough driver part*, as on figure A-2. The driver maps HPA ranges of the device registers to GPA ranges used by the VM. The *Virtual Machine* (VM) cannot do it by itself as a user-space process, only the passthrough driver (a host OS module) has the visibility of HPA ranges assigned to the registers of the device.

If a device instance, selected previously by the GSO, is already attached to the computing node the GSO may decide to reuse the same `upa_c_devctl` and `upa_c_iommu` registers and only assign another pair of `hpa_c_devctl2` and `hpa_c_iommu2` to them. Registers operations will not conflict at device side, provided that they are tagged by the `VM_GID` – a globally unique guest identifier, used also to differentiate the IOMMU translation maps.

Once the disaggregated device is attached by GSO, the passthrough driver in the host can complete its configuration.

First, using the `hpa_c_iommu`, the driver sends a list of GPA ranges describing guest's RAM in order to complete the IOMMU translation maps building (initialization men-

tioned in a previous section) for a given VM. This allows the remote device to receive GPA pointers as parameters of an operation request, and, with the help of the DMA engine and the IOMMU, to fetch data from the disaggregated guest RAM.
Then, the remote interrupt, which routing to the local IC was already configured, is properly set to be received by the VM.
Next, the control registers of the device, attached to the host as `hpa_c_devctl` are mapped to the VM as `gpa_c_devctl`. From this moment on, the guest driver can operate the device as if it was available locally.
Finally, marked as a *VM part* on figure A-2, the VM configures a memory resize notifier to notify the passthrough driver about all guest RAM extension or shrinking events. After the direct attachment is configured, this is necessary in order to keep the IOMMU translations maps up to date before each operation. Otherwise, the guest could ask the device for processing data which the IOMMU is not aware of, or worse, a malicious workload could use outdated mappings to access ranges of disaggregated memory that are no longer used by a given VM. It should be noted that the IOMMU provides a good memory protection as long as it is properly configured.

Sharing a device between VMs

If a device can emit only one interrupt but the device can be attached by multiple VMs, there is an additional serialization needed. It is up to the logic of the computing node's *interconnect interface* to interleave processing requests in a *transactional* manner. A transaction starts when an operation is triggered and lasts until a completion interrupt arrives. It is only after an interrupt has been correctly redirected that another transaction can be processed. Interleaving processing requests in a way that ignores interrupts is not allowed because to which VM an interrupt should be dispatched would not be known. The interrupt would thus be injected to all guests sharing a device and all but one would eventually ignore it, although their execution would still be disturbed – this is suboptimal.
With such a transactional shared configuration the requirements of a *direct attachment* mode are still met, but of course the device's throughput will be shared between all participating VMs. This type of configuration is also often referred to as a *mediated*

passthrough. Sharing a device in this way can be especially useful for peripherals that do not support hardware multiplexing, for example non SR-IOV peripherals.

Design comments

With a given heterogeneous system architecture the presented approach seems to be the best analogy to the passthrough attachment in its traditional shape.
Alternatively, in order to rule out the GSO role, a device would need to have an access to the HPA ranges of the computing node. For example it would need to be attached to the PCIe bus either directly or through a PCIe switch. Nevertheless, it is not the same level of disaggregation if a device is not attached to a fully independent system node.
Furthermore, the design could eliminate the IOMMU programming but then a VM would need to intercept each device access to capture the GPA addresses generated by the guest device driver. Subsequently, they would need to be translated to HPA addresses of the computing node and further to corresponding UPA ranges of the memory node providing guest's RAM resources. The latter, however, can be only done by the GSO, which means that a central system component would need to be involved in each device transaction. This is a bad design choice as the GSO would very likely become a scalability bottleneck. Except for that, intercepting each device registers access is against the principles of a direct attachment configuration.

A.2 Infrastructure setup in more details

This section expands Section , in which many details were intentionally omitted for better readability.
The idea is that the passthrough driver, on behalf of a VM, sends only one request to the GSO and receives back one response as an operation result. This is a clear approach, because the attachment can only succeed or fail; from a VM point of view it cannot be partially completed. It is a GSO responsibility to perform a sequence of configuration steps and, in case one of them fails, to rollback the previous ones in order to return to a clean setup.
The device attachment configuration is presented in more details on figure A-3. It is

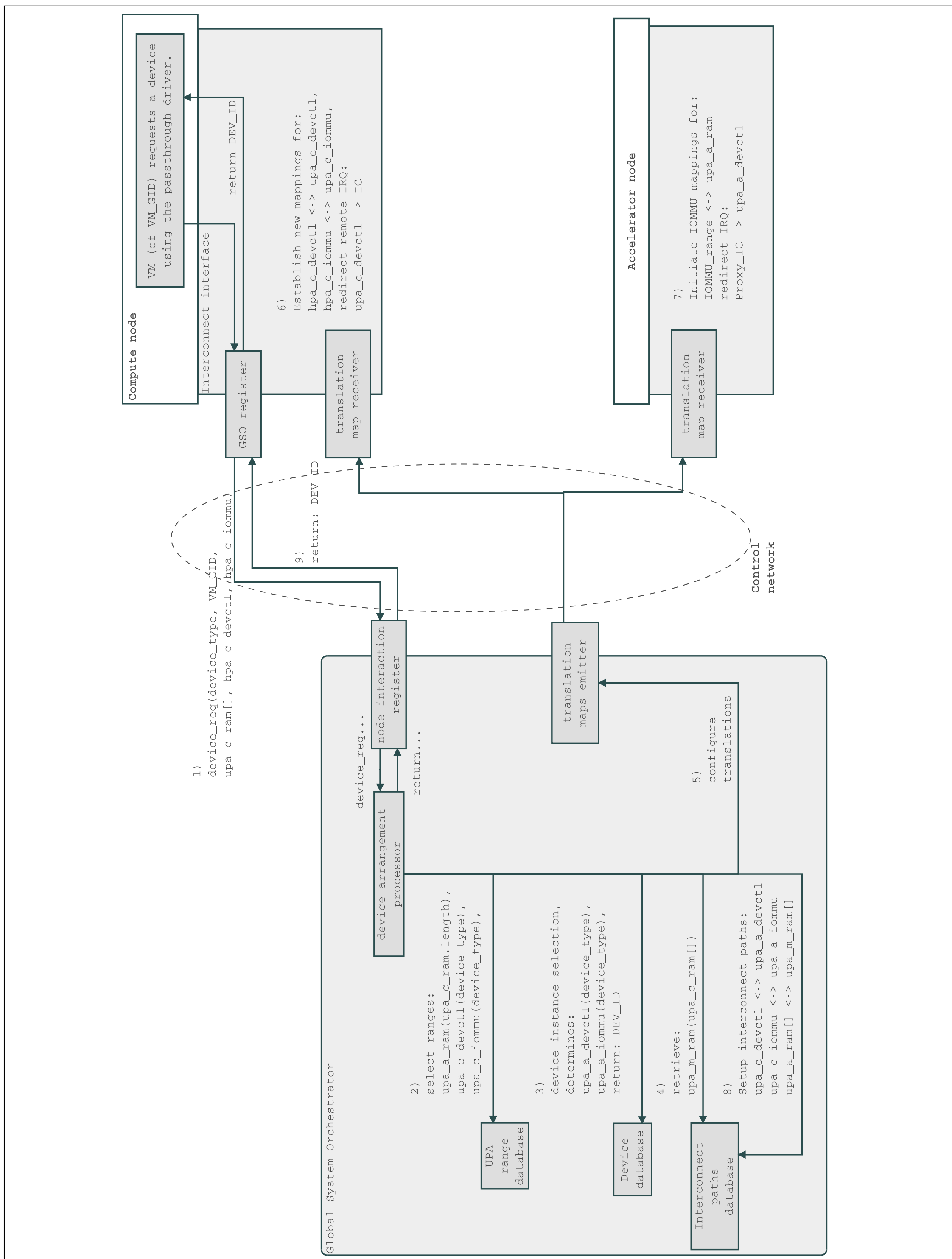


Figure A-3: Device attachment configuration at GSO side

also useful to refer to figure A-1 as a complementary illustration.

Both the request and the response provide additional parameters:

device_type : A flag specifying the kind of device that the guest asks for. This assumes that flags are assigned up to a common system convention.

VM_GID : A globally unique guest identifier, provided by the GSO at VM boot time, used to associate reserved resources with a VM in the GSO registers and to differentiate IOMMU mappings, which are guest-specific.

upa_c_ram[] : A list of UPA ranges associated with the remote RAM of the guest. Although the driver can directly see only the corresponding `hpa_c_ram[]` visible by the host, we assume that it has a way to query the related UPA ranges at the local *interconnect interface*. Otherwise, this translation needs to be done by the GSO, which would need to keep track of each binding of each VM. This is an additional bulk of information to store and process at the central system node. Thus the latter approach is better avoided.

hpa_c_devctl : A reserved HPA region representing the device control registers, to be mapped by the GSO with the `upa_c_devctl` during the attachment process. Its size depends on the device type.

hpa_c_iommu : A reserved HPA region representing the remote IOMMU unit, to be mapped by the GSO with the `upa_c_iommu` during the attachment process. Its size may also depend on the device type.

Upon receiving a request, the execution moves to step `2)` on figure A-3. The GSO selects new UPA ranges that will act as end-points of interconnect paths created at later stages: the `upa_c_devctl` and `upa_c_iommu` ranges discussed above, plus the `upa_a_ram` allotted to the device node. The last range will allow the IOMMU to reach the guest's RAM on a remote memory node and addresses the challenge **Ch.2**. These ranges are selected on-demand because which VM will need which device is not known. Except for established connectivity (done at the end) the challenge **Ch.1** is addressed at this point.

In the most generic case, upa_a_ram mirrors the upa_c_ram, which means that the whole guest's RAM is available for the device. Nevertheless, it is possible that only a subrange of it should become available for a particular device, for example when it is possible to narrow down the scope that will contain input data.

Subsequently, in step 3), a particular instance of the requested device_type is selected, according to the internal policies of the GSO. For example, it may be preferable to involve a device node already used by other workloads, in order to keep a minimum number of device nodes powered-on. Or oppositely, a guest may prefer to attach a device exclusively, which is still possible, although a general idea is to share installed peripherals as much as possible. This is up to a workload-specific deployment policy, not the mechanism itself. The device instance selection automatically determines upa_a_devctl and upa_a_iommu, which were assigned to it when it was registered to the system. Another outcome of this step is the DEV_ID, a globally unique identifier of the selected instance attachment, which will be returned to the VM when the operation completes.

Having the upa_a_ram reserved, as a step 4), the GSO must figure out the upa_m_ram range to be paired with the first one. It was not directly received with the driver request, which has no visibility of it. Instead, the driver can provide the upa_c_ram and the corresponding upa_m_ram is looked-up in a registry of existing interconnect links.

At this point all pieces of information required to configure the *interconnect interfaces* at the computing node and at the device node are collected. In step 6) the computing node receives new mappings for the device control registers and the associated IOMMU registers (hpa_c_devctl: upa_c_devctl and hpa_c_iommu: upa_c_iommu, respectively). A remote device interrupt line (subrange of the upa_c_devctl) is attached to the local *Interrupt Controller* (IC).

The device node is programmed in step 7). A subset of the IOMMU registers is already mapped to upa_a_iommu. It was statically assigned when the node was registered to the system because this UPA range is a fixed part of the *interconnect interface*. Nevertheless, the rest of the IOMMU address range is built dynamically by VM-specific translation maps building. The IOMMU is supposed to translate the GPA addresses used by the guest driver (**input column** of a map), to UPA addresses

from the upa_a_ram range (**output column** of a map). The GSO initiates building the translation map by passing the upa_a_ram ranges, which stand for the **output column**. The input column will be provided later by the driver.

There is a way to make sure that entries of both columns will be properly matched, even if created by two different system components. A guest RAM is already known and can be expressed, in general, as a list of contiguous ranges, with each range described by a pair of *(offset, length)*. For a given guest, the offsets may be different in different address spaces (gpa_c_ram vs. upa_a_ram) but lengths are equal. Therefore, by keeping a consistent list ordering, different components are able to build the translation map correctly.

Example: Guest's RAM is reachable through a list of ranges:

```
gpa_c_ram = [(gcr_offset1, len1), (gcr_offset2, len2)]
```

Therefore, in the UPA, it is expressed as:

```
upa_c_ram = [(ucr_offset1, len1), (ucr_offset2, len2)]
```

During the UPA ranges reservation for a device node, a corresponding list is:

```
upa_a_ram = [(uar_offset1, len1), (uar_offset2, len2)]
```

It may happen that uar_offset1 > uar_offset2 but it does not matter. It is important that corresponding ranges have the same positional index in the list. Subsequently, when building the IOMMU translation map, the GSO provides the upa_a_ram in a given order for an output column. After that the guest has to provide the gpa_c_ram as an **input column** in the same order, to keep the mapping correct.

Having it done, the challenge **Ch.3** is solved.

In addition to guest RAM access configuration, the Proxy IC module (pretending a standard IC from a device perspective) has to be configured. Interrupts generated by the device and destined for a configured guest are redirected to a dedicated subrange of the upa_a_devctl, which pairs up with the upa_c_devctl side. This step solves the challenge **Ch.4**.

Finally, in step 8), the GSO must establish new interconnect paths in order to enable the communication between the following ranges:

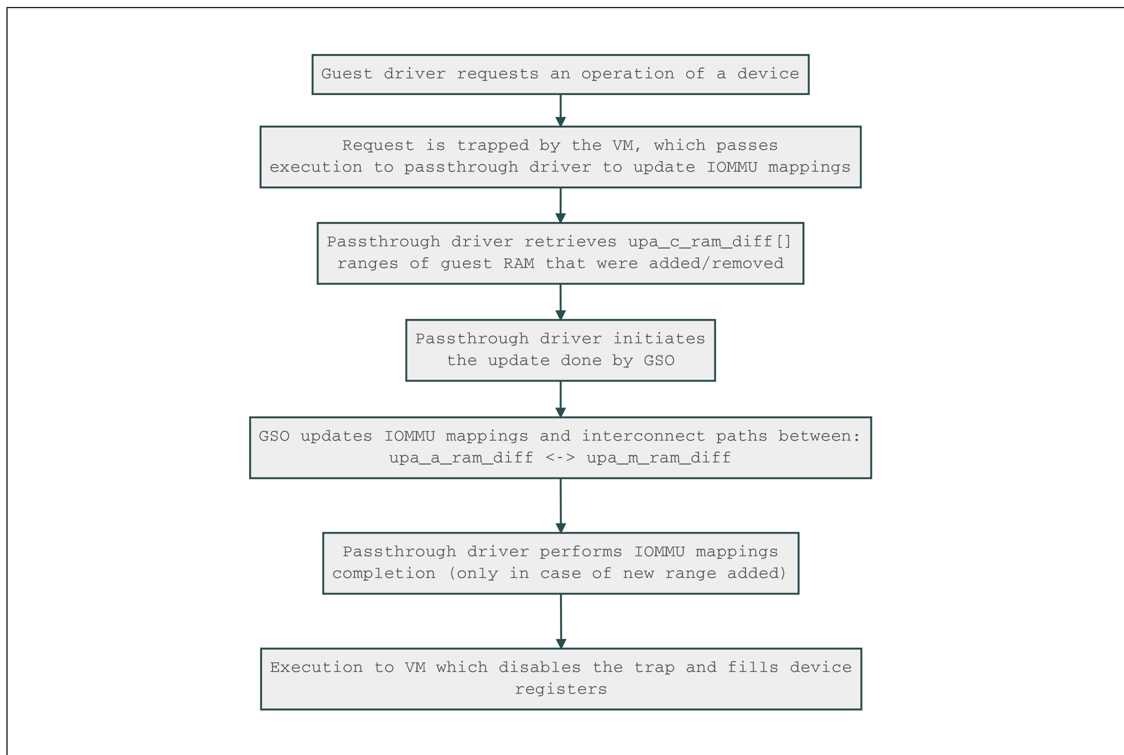


Figure A-4: Updating directly attached device after guest memory resize

```
upa_c_devctl <-> upa_a_devctl
upa_c_iommu  <-> upa_a_iommu
upa_a_ram    <-> upa_m_ram
```

Eventually, in step `9)`, the `DEV_ID` is returned to the guest and it is used to complete the configuration, as well as any further reconfigurations.

A.3 IOMMU maps update on guest RAM resize

As mentioned at the end of section , in the last part of direct device attachment configuration a VM installs a memory resize notifier in order to detected when a guest RAM is enlarged or shrank. This is necessary to keep the IOMMU address translation mapping always up to date; this process is explained in this section.

Figure A-4 illustrates the workflow performed for each device attached in passthrough mode during an update of the memory mappings.

Normally, if no guest RAM resize happened since the last configuration update, a processing request issued by the guest driver should be directly transferred to the remote device. Otherwise, once a VM is notified about the resize, it sets a trap on the

gpa_c_devctl register access, so that the next processing request is intercepted and the IOMMU updated beforehand. For example, if there are multiple RAM resizes between two processing requests, it could be better to batch them together and update the IOMMU mappings in a *lazy* manner, that is, just before the next operation. Chances are that a memory section was added to handle a short-time burst and removed just after, so no actual update is necessary from the IOMMU perspective. After the update, the configuration is considered *clean* and the trap is disabled, until next resize.

On the other hand, if a device processing latency must be invariant and predictable, for example in case of real-time operations, a completely opposite approach can be preferred and the update of the IOMMU mappings can be performed immediately after each RAM resize.

The steps of a configuration update are a subset of those performed during attachment since only memory mappings are changed, control registers and interrupt redirections are not affected. First, as figure A-4 shows, the passthrough driver collects the upa_c_ram_diff[] array describing the ranges of disaggregated guest RAM that were added or removed since the last device configuration. Then, most of the modification is done by the GSO. It is only in case of a guest's memory extension that the passthrough driver has to complete the corresponding IOMMU mappings at the end, analogously as described in A.2 for gpa_c_ram[] (now the new guest memory would be expressed as gpa_c_ram_diff[]).

The part of reconfiguration done by the GSO is presented in more details on figure A-5. The GSO is the only component capable of computing the upa_m_ram_diff and reserving the corresponding upa_a_ram_diff ranges, based on upa_c_ram_diff passed by the passthrough driver. Then, the IOMMU mappings of the device node are updated and the proper interconnect paths are created or destroyed, depending on the type of update.

A.4 Disaggregated device detachment

After describing in details how a disaggregated device can be attached to a VM and how its configuration supports runtime guest memory RAM resizing, we conclude

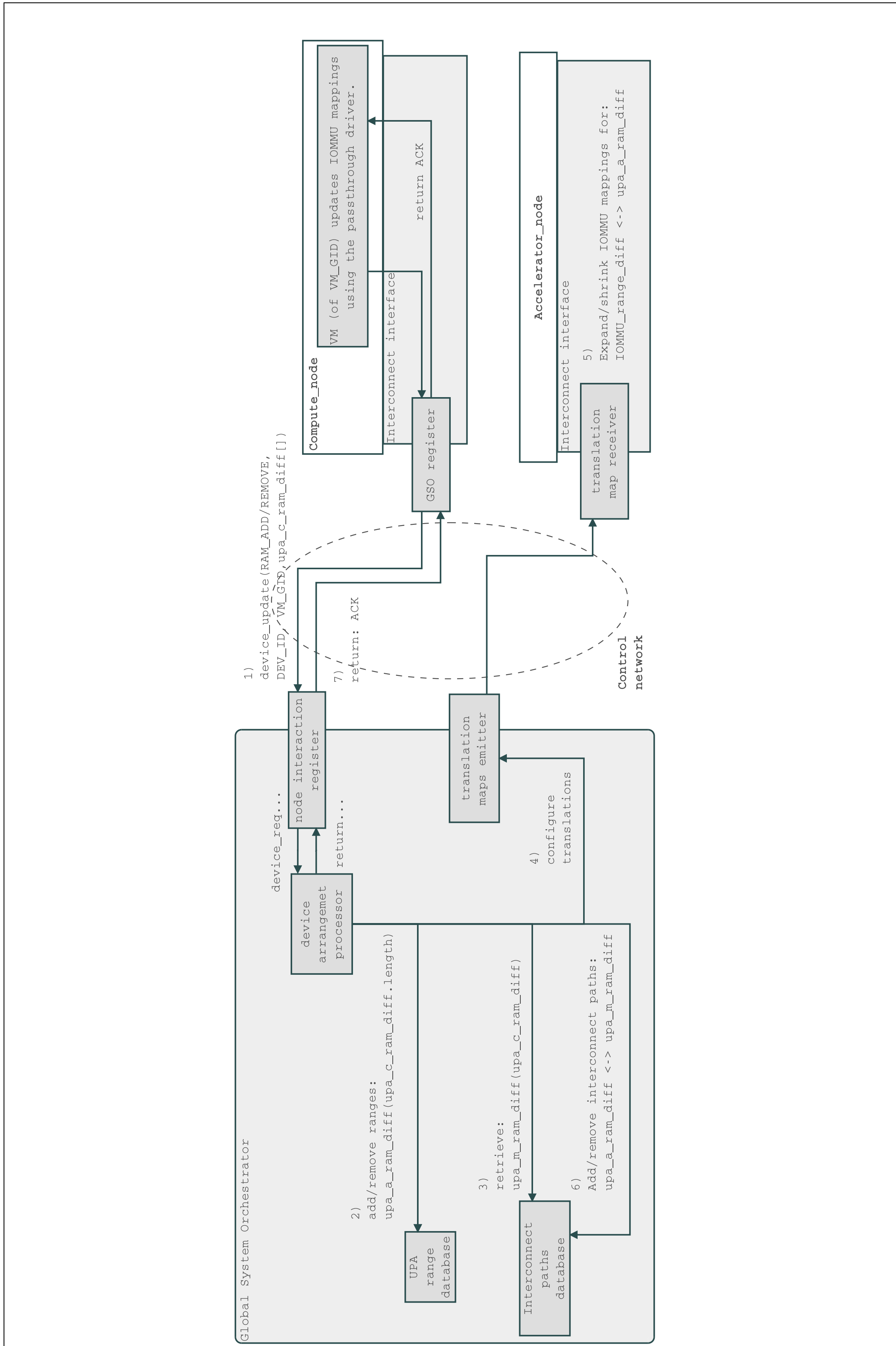


Figure A-5: Updating directly attached device after — GSO part

this discussion with the detachment of a disaggregated device. Eventually, when a disaggregated device is going to be detached, all associated resources must be cleaned up at the device node (IOMMU translation maps, device-internal setup), in the GSO logic (release of UPA ranges, cleanup of interconnect paths) and at the *computing node* (interrupt reception, registers mapping to GPA). Upon detachment the sequence of steps is actually just the reverse of the sequence described in Section .
A device may be detached for several reasons:

- For example, a VM may voluntarily decide to release a device, knowing that it will not need it anymore. Assuming that a single VM is running a single workload, this situation can occur when the deployed application needs the device only temporarily and releases it in an explicit way.

- Similarly, a VM may simply terminate. The device detachment would then be one of the steps of an overall resources cleanup.

- Finally, as any other piece of software, deployed workloads may occasionally crash and it can affect a VM state in such a way that it needs to be restarted. The detachment must then be conducted without any guest cooperation as the guest can be inoperable. Note that in theory, VMs may also get into erroneous state that would require reboot. However, an industry level system should be characterized by a high stability and such a scenario should remain unlikely.

ÉCOLE DOCTORALE
universite PARIS-SACLAY
Sciences et technologies de l'information et de la communication (STIC)

Titre: Nouvelles techniques de virtualisation de la mémoire et des entrées-sorties vers les périphériques pour les prochaines générations de centres de traitement de données basés sur des équipements répartis déstructurés

Mots clés: virtualisation, mémoire, systèmes désagrégées

Résumé: Cette thèse s'inscrit dans le contexte de la désagrégation des systèmes informatiques - une approche novatrice qui devrait gagner en popularité dans le secteur des centres de données. A la différence des systèmes traditionnels en grappes, où les ressources sont fournies par une ou plusieurs machines, dans les systèmes désagrégés les ressources sont fournies par des nœuds discrets, chaque nœud ne fournissant qu'un seul type de ressources (unités centrales de calcul, mémoire, périphériques). Au lieu du terme de machine, le terme de créneau (slot) est utilisé pour décrire une unité de déploiement de charge de travail. L'emplacement est assemblé dynamiquement avant un déploiement de charge de travail par l'orchestrateur système.
Dans l'introduction nous abordons le sujet de la désagrégation et en présentons les avantages par rapport aux architectures en grappes. Nous ajoutons également au tableau une couche de virtualisation car il s'agit d'un élément crucial des centres de données. La virtualisation fournit une isolation entre les charges de travail déployées et un partitionnement flexible des ressources. Elle doit cependant être adaptée afin de tirer pleinement parti de la désagrégation. C'est pourquoi les principales contributions de ce travail se concentrent sur la prise en charge de la couche de virtualisation pour la mémoire désagrégée et la mise à disposition des périphériques.
La première contribution principale présente les modifications de la pile logicielle liées au redimensionnement flexible de la mémoire d'une machine virtuelle (VM). Elles permettent d'ajuster la quantité de RAM hébergée (c'est à dire utilisée par la charge de travail en cours d'exécution dans une VM) pendant l'exécution avec une granularité d'une section mémoire. Du point de vue du logiciel il est transparent que la RAM proviennent de banques de mémoire locales ou distantes.
La deuxième contribution discute des notions de partage de mémoire entre machines virtuelles et de migration des machines virtuelles dans le contexte de la désagrégation. Nous présentons d'abord comment des régions de mémoire désagrégées peuvent être partagées entre des machines virtuelles fonctionnant sur différents nœuds. De plus, nous discutons des différentes variantes de la méthode de sérialisation des accès simultanés. Nous expliquons ensuite que la notion de migration de VM a acquis une double signification avec la désagrégation. En raison de la désagrégation des ressources, une charge de travail est associée au minimum à un nœud de calcul et a un nœud mémoire. Il est donc possible qu'elle puisse être migrée vers des nœuds de calcul différents tout en continuant à utiliser la même mémoire, ou l'inverse. Nous discutons des deux cas et décrivons comment cela peut ouvrir de nouvelles opportunités pour la consolidation des serveurs.
La dernière contribution de cette thèse est liée à la virtualisation des périphériques désagrégés. Partant de l'hypothèse que la désagrégation de l'architecture apporte de nombreux effets positifs en général, nous expliquons pourquoi elle n'est pas immédiatement compatible avec la technique d'attachement direct, est pourtant très populaire pour sa performance quasi native. Pour remédier à cette limitation, nous présentons une solution qui adapte le concept d'attachement direct à la désagrégation de l'architecture. Grâce à cette solution, les dispositifs désagrégés peuvent être directement attachés aux machines virtuelles, comme s'ils étaient branchés localement. De plus, l'OS hébergé, pour lequel la configuration de l'infrastructure sous-jacente n'est pas visible, n'est pas lui-même concerné par les modifications introduites.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

ÉCOLE DOCTORALE

universite PARIS-SACLAY

Sciences et technologies de l'information et de la communication (STIC)

Title: Novel memory and I/O virtualization techniques for next generation data-centers based on disaggregated hardware.

Keywords: virtualization, memory, disaggregated systems

Abstract: This dissertation is positioned in the context of the system disaggregation - a novel approach expected to gain popularity in the data center sector. In traditional clustered systems resources are provided by one or multiple machines. Differently to that, in disaggregated systems resources are provided by discrete nodes, each node providing only one type of resources (CPUs, memory and peripherals). Instead of a machine, the term of a slot is used to describe a workload deployment unit. The slot is dynamically assembled before a workload deployment by the unit called system orchestrator.
In the introduction of this work, we discuss the subject of disaggregation and present its benefits, compared to clustered architectures. We also add a virtualization layer to the picture as it is a crucial part of data center systems. It provides an isolation between deployed workloads and a flexible resources partitioning. However, the virtualization layer needs to be adapted in order to take full advantage of disaggregation. Thus, the main contributions of this work are focused on the virtualization layer support for disaggregated memory and devices provisioning.
The first main contribution presents the software stack modifications related to flexible resizing of a virtual machine (VM) memory. They allow to adjust the amount of guest (running in a VM) RAM at runtime on a memory section granularity. From the software perspective it is transparent whether they come from local or remote memory banks.
As a second main contribution we discuss the notions of inter-VM memory sharing and VM migration in the disaggregation context. We first present how regions of disaggregated memory can be shared between VMs running on different nodes. This sharing is performed in a way that involved guests which are not aware of the fact that they are co-located on the same computing node or not. Additionally, we discuss different flavors of concurrent accesses serialization methods. We then explain how the VM migration term gained a twofold meaning. Because of resources disaggregation, a workload is associated to at least one computing node and one memory node. It is therefore possible that it is migrated to a different computing node and keeps using the same memory, or the opposite. We discuss both cases and describe how this can open new opportunities for server consolidation.
The last main contribution of this dissertation is related to disaggregated peripherals virtualization. Starting from the assumption that the architecture disaggregation brings many positive effects in general, we explain why it breaks the passthrough peripheral attachment technique (also known as a direct attachment), which is very popular for its near-native performance. To address this limitation we present a design that adapts the passthrough attachment concept to the architecture disaggregation. By this novel design, disaggregated devices can be directly attached to VMs, as if they were plugged locally. Moreover, all modifications do not involve the guest OS itself, for which the setup of the underlying infrastructure is not visible.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France