Dynamic guest memory resizing — paravirtualized approach

Maciej Bielski, Alvise Rigo
Virtual Open System
{*m.bielski, a.rigo*}*@virtualopensystems.com*

Renaud Pacalet
Télécom ParisTech, Université Paris-Saclay, LTCI
renaud.pacalet@telecom-paristech.fr

***Abstract*—Nowadays cloud-computing systems take a great advantage of virtualization for the benefits of workload isolation and flexible resources partitioning. It is expected that the same functionalities will be available also on disaggregated architectures, proposed recently as next generation approach for building data-centers. In this publication, we are presenting the design and prototype of an enhanced virtualization layer, enabling runtime memory balancing between virtual machines on a section granularity. Guests' RAM is backed by isolated chunks of host memory, coming from independent physical banks, not necessarily a local one. It can be dynamically resized without requiring any support for ACPI emulation in the virtualization framework, as we exemplified by implementing the prototype on ARMv8 platform.**

***Keywords*-memory balancing, virtualization, cloud computing, memory disaggregation**

I. Introduction

In today's cloud computing systems virtualization layer unquestionably plays an essential role. It comprises functionalities provided by the host OS (optionally supported by user-space tools), necessary to run virtual machines (VM) executing guest operating systems. Virtualization abstracts and partitions underlying resources making them accessible to the user conveniently, without the need of knowing neither the physical system architecture nor the components' actual location. Multiple VMs can coexist together on the same host and isolate different workloads, increasing overall system security and resources utilization efficiency.

In this work we present a design and prototype implementation of an enhanced virtualization layer enabling runtime adjustment of the guest memory without relying on the ACPI standard [4] support. The support may either not be present at all [2] or it may rely on a closed proprietary firmware, which we preferred to avoid for potential security concerns. Thanks to runtime VM memory resize there is no service downtime, which could likely happen if a workload was migrated to other VM, equipped with more RAM. It also allows to keep all memory resources maximally used all the time by easier balancing them between different VMs.

Our prototype is based on open-source software, both host and guest are running *Linux* kernel and virtualization capabilities are provided by the QEMU-KVM [6], [8] framework. Passing information between host and guest is being performed in a paravirtualized manner. 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. QEMU is a complementary host user-space software, capable of taking advantage of KVM, handling a VM emulation and executing a guest OS.

Paravirtualization, in general, is a technique especially popular for guest device drivers when the driver is split into two parts; front-end installed in the guest and back-end installed in the host. The former exposes a standard interface expected by guest user-space and communicates with the latter, part of the hypervisor, that eventually performs an intended task, for example operates underlying hardware.

What is a crucial trait of the presented approach, the guest memory buffers are not obtained from the standard host memory allocator (e.g. through *malloc()* function) but instead from a custom driver managing isolated ranges of *host physical address space* (HPA). The driver abstracts the actual memory location, it can come from locally available resources or from a disaggregated memory pool. Disaggregation is an architectural approach that, in the memory context, means that only a limited amount of RAM may be locally available while most of it is physically decoupled from the host CPU cores and come from a remote system node [1], [15], [16], [19]. Assuming that partitions of remote memory could be dynamically attached and detached, we propose a method of exposing them to virtual machines from the host level as isolated contiguous ranges of HPA. However, for the lack of a real disaggregated hardware at the time of writing, our solution is prototyped on a system with all memory resources available locally.

The rest of the paper is structured as follows. Section II presents several related works, Section III details the software architecture we propose, Section IV presents the testbed we implemented for evaluation and obtained results, Section V concludes the paper.

II. Related work

Most prior works related to guest memory balancing implemented different flavors of page-based techniques, based on the *overcommitment* approach. They are based on juggling physical memory pages between the larger set of virtual pages to back-up the overcommitted ones.

The most straight-forward approach is the *uncooperative swapping*. Since performed by hypervisor, it suffers from a so called *semantic gap*, meaning the hypervisor cannot optimally select guest pages to be evicted for the lack of guest memory usage [9], [10].

Ballooning has been proposed to address this problem by delegating the page selection process to the guest [17]. Nevertheless, a VM that needs more pages relies on cooperation with other VMs while they can limit or disable *ballooning*. Moreover, it is slower than *uncooperative swapping* due to possible page evictions in other guests [11]. Combinations of

both mechanisms have been proposed to select best candidates and improve balancing responsiveness [11], [12]. Even so, page-base techniques operate within the memory amount defined at boot time and cannot expand it. Moreover, they fragment guest memory for the lack of control over physical addresses of pages (decided by the memory allocator) [13], [14].

Instead of *swapping*, we employed *mapping*-based method. It allows to extend guest's memory (by performing the *hot-plug* operation) over an initially declared value by attaching a section of RAM provided by a custom host driver. In case a section needs to be detached, there is always only one VM responsible of releasing it. Such approach simplifies sections management between multiple guests and does not compound host's memory fragmentation.

Up to authors' best knowledge, only one paper by Liu et al. presents a hybrid system combining *ballooning* together with *memory hot-plug* mechanism to overcome the limitation of guest memory upper-bound [13]. The crucial difference is that, this work uses the default host memory allocator, while in our system the VM memory buffers are obtained from isolated chunks and are physically contiguous. We consider this design to be easily adaptable with disaggregated systems, where memory resources come from different remote system nodes. Moreover, we don't integrate ballooning, in its classic flavor, for the reason that it may render a memory section impossible to unplug and the necessary migration would impose additional performance overhead. Our 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 other difficulties mentioned above.

With regards to physical memory detection, in our prototype the guest OS relies on information from provided *device-tree* structure at boot time and respective runtime communication held in a paravirtualized manner. Alternatively, in both cases it could be based on the ACPI standard [4] but this requires respective firmware emulation to be provided by virtualization framework and corresponding support in the operating system.

III. Proposed system architecture

The virtualization layer enhancements span three levels.

The first one is a custom host OS (hypervisor) driver that initializes data structures related to isolated memory. The isolated memory is indicated at boot time by the *device-tree*, that is a description of a hardware configuration, typically used to inform *Linux* kernel of available resources and their parameters in order to initialize underlying hardware properly. For isolated RAM the initialization process is very similar to regular one but corresponding ranges of HPA are not eventually passed to be managed by the system allocator. Instead, this memory will be available for the VMs only through the driver.

The second level is a VM itself. Similarly to regular processes, each VM instance resides in the host local RAM but all memory buffers obtained to constitute guest memory are allocated from the isolated memory pool. Also, while a guest is running, a VM is capable of expanding and shrinking its memory, which operation is called the *guest-physical memory* resize. On top of that, a VM interacts with the guest OS for receiving resize requests as well as to coordinate respective *logical memory* resize at guest side.

The guest OS is a third level of modifications. It exchanges messages with the VM and adapts the guest's *logical memory* by performing *hot-add/-remove* operations.

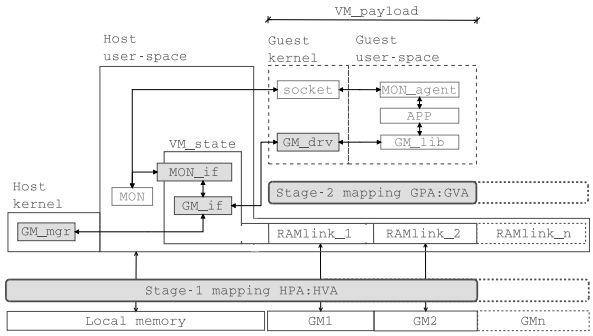

Figure 1: Guest memory provisioning from isolated pool

Figure 1 presents a more detailed system view. The core components and their roles are described further in this section. Others are mentioned in the following part and may or may not be needed, depending on a particular communication path between a workload and the host.

- `GM`k: Guest memory *backends* — chunks of physically contiguous memory isolated from the host allocator and used exclusively to build guest RAM, $k \in [1, 2, \ldots, n]$.
- `RAMlink_`k: VM-internal data structures abstracting guest RAM resources, during initialization mapped *1-to-1* to `GM`k by the host driver `GM_mgr`, $k \in [1, 2, \ldots, n]$.
- `GM_mgr`: Guest memory manager implemented as a host driver. Upon request from a VM, it selects a `GM`k region and maps it to VM process address space, specifically to `RAMlink_k` range.
- `VM_state`: A set of data structures and operations typically performed by a VM, including a guest system execution management.
- `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).
- `GM_drv`: Guest-side driver, responsible for communication with the VM and triggering the *logical memory resize*.
- `GM_if`: A VM communication interface, interacts with several other components during the memory resize:
  - Host driver (`GM_mgr`), to requests for binding a `RAMlink_`k instance with an actual host memory chunk `GM`k.
  - Guest driver `GM_drv` to receive memory resize requests directly from guest OS or interact with the VM to ask for physical RAM attachment or detachment.

– VM monitor interface (MON_if) to receive memory resize request (alternative way) signaled by external processes.

The manager module (GM_mgr) is responsible for selecting a range of isolated memory (GM*k*) that will be linked to a given VM by a RAMlink_*l*. This linkage happens either when a new VM is launched or at runtime, when its RAM volume is resized.

The first case is more straightforward as resources are properly initialized at host side and from the guest's perspective they are statically predefined in 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) of obtaining resources from host memory allocator, a VM (GM_if) uses a specific driver (GM_mgr) to reserve one or multiple *backends* (GM*k*). Therefore, guest RAM is compounded by one or multiple chunks of isolated RAM.

In case of dynamic memory resize there are additional steps 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 both are discussed in sections III-B and III-C, respectively. Additionally, respective runtime modifications are needed at guest side in order to make new memory resources available for guest processes, or oppositely to properly stop using these, which 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, that is a physically contiguous range of memory of a platform-specific size (for example 1 GB).

Whether it is a guest or a VM, that performs the reconfiguration first, depends on an operation type. In the event of guest RAM expansion, firstly new *guest physical* memory resources have to become available before guest can start using them. Symmetrically, in case of RAM shrinking, the guest has to stop using involved *sections* first, before a VM can safely release corresponding memory *backends*. Figure 2 presents a flow diagram of this operation in both cases, triggered by an external process through the monitor interface MON_if.

A. Resize volume

The incoming resize request specifies a desired amount of VM memory, which, after comparing it against currently available amount, allows to determine whether a VM RAM should be extended (positive difference) or shrunk (negative difference) and an absolute value of the computed difference needs to be expressed as a multiple of size of one backend. 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. To the contrary, when shrinking the memory, one can assume that the workload is ready to execute with **no less than** indicated amount of RAM, thus the absolute value of a difference is rounded down.

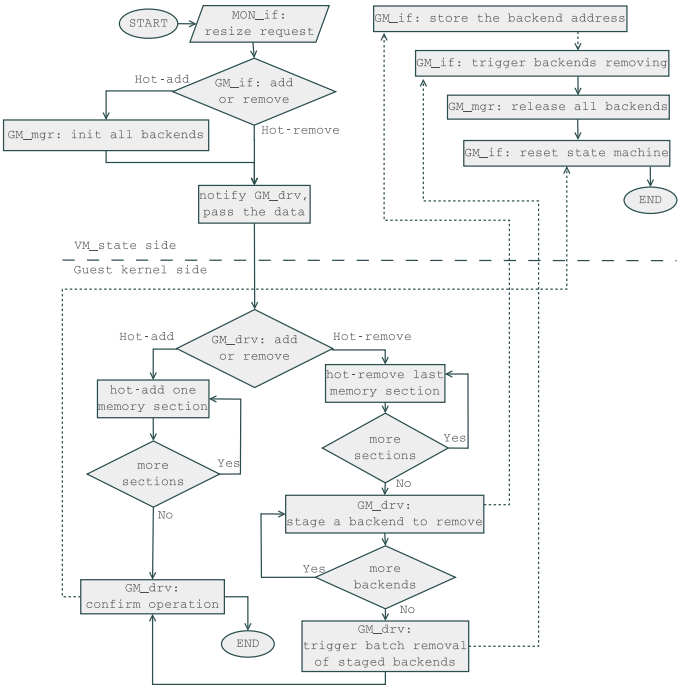

Figure 2: Dynamic memory resize

As already mentioned, the guest *memory section* size is a platform-specific parameter of the *Linux* kernel. Since each initialized *section* has to be associated to a VM *backend*, the size of a backend should equal to one or multiple of a *section* size, while the lower the VM *backend* size, the better RAM resizing granularity. Therefore, the case of a VM *backend* size equal to guest *memory section* size seems to be optimal.

B. Live VM balancing: guest parameters visibility

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

Observing only a VM state (as oppose to considering also guest-internal parameters) has the advantage of no requirements towards software installed on the guest but can only provide memory consumption data of limited accuracy.

More precise data can be collected directly from a running guest, however this requires a dedicated service (depicted as MON_agent on figure 1) to maintain the communication and query proper parameters. Such 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. In this scenario, the host VM manager (MON) could collect more precise data (e.g. regular memory vs. *page cache* consumption or amount swapped memory) and resize guest RAM when needed.

C. Explicit resize requests

Orthogonally to live VMs balancing, a RAM resize request could be also emitted by the guest itself and there are two possible categories: *pro-active* and *reactive*, with respect to the moment when the guest runs out of memory.

The first one assumes that this situation could be avoided since the need for additional RAM can be anticipated. In one scenario, a guest component (`MON_agent`) performs continuous monitoring of relevant parameters and emits a resize request when needed. Again, this method does not significantly involve a workload owner, which is good, especially for legacy applications that cannot be adapted to the system. The drawback of this approach is that a continuous monitoring could imply a significant performance overhead [9], [18]. In other scenario, instead of being a standalone guest component, the `MON_agent` would be integrated with a user workload (`APP` on the figure 1). This requires that an application can notify the environment in advance about a need for more memory or when some of attached sections can be detached. This approach is much more precise and lightweight as the application knows exactly when and how much more resources will be needed.

The second category is conceptually the most straightforward, as no guest monitoring or application integration is performed. Instead, the guest is allowed to run *out-of-memory*, and ask for more resources before handling the situation by taking standard steps like page swapping or kiling other processes. One problem is that the process of RAM expanding needs a certain amount of memory itself, whereas the system just exceeded all its resources. Therefore, there needs to be a buffer reserved a priori for each potential section to serve such situation. Another drawback is that this method can only attach more resources. Therefore, the *reactive* approach seems more like a last resort solution in case *pro-active* techniques would not be sufficient.

D. Request path

The last element of the figure 1, shortly mentioned before, is the `GM_lib`, exemplifying an alternative communication path between a workload and the host environment. Instead of using 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 the operation. Regarding the figure 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 a back-end while the `GM_drv` plays a front-end role.

E. Resize granularity

The presented system provides a *section-based* technique for dynamic memory balancing between VMs running on the same host. This is different than traditional *page-based* mechanisms like memory *ballooning* [17], which operates within the scope of VM memory defined at boot time. Our approach aims to keep the runtime flexibility but avoid 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 be constrained by a launch-time declared maximum value. The price of such 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. Moreover, since requirements of applications are typically changing over time, operating at the coarser granularity limits the control signals traffic intensity between host and VMs required otherwise to pass page ownership to other guest.

F. Disaggregation context

A resizeable memory virtualization layer based on isolated *backends*, as above, is ready to be adapted for disaggregated architectures. The `GMk` *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 section granularity assumed for this work seems to be even more reasonable as the amount of globally available memory may be in the range of terabytes.

IV. Implementation and evaluation

A. Implementation details

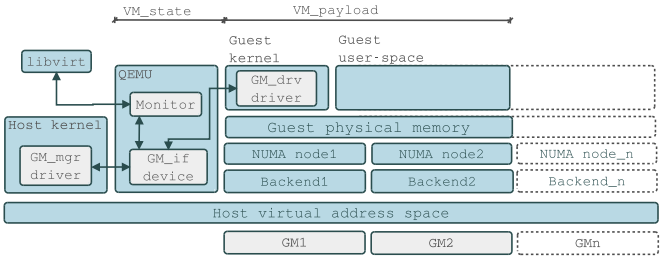

Figure 3: System prototype components

This section briefly describes the system prototype with components illustrated on the figure 3. The work was conducted within the *dReDBox* project, which defined the target platform to be ARMv8. Nevertheless, except for parts of memory *hot-plug* functionality, the implementation is platform-agnostic. The system is based on open-source software, *Linux* running both as host and guest OS and *QEMU* being the host user-space process that communicates with KVM driver in the host and handles VM emulation. 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. This mapping is performed by the `GM_mgr` host driver, that based on information from the *device-tree*, initializes subsequent `GMk` ($k \in [1, 2, \ldots n]$) HPA ranges as a custom flavor of *reserved memory* (in *Linux* kernel parlance), which are not passed to the memory allocator. This effectively provides memory isolation, since the access can be granted only by the `GM_mgr` driver. On *QEMU* side, the

crucial modifications developed for the system prototype are the following:

1) A custom backend created for *non-uniform memory access* (NUMA) abstraction that we reused. This allows to construct resizeable guest memory out of preallocated buffers.Instances of the backend (`RAMlink_`k) acquire resources from `GM_mgr`, which maps `GM_`k ranges to 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 *telnet* interface (`MON_if`) and often used by external tools (`MON`, for example *libvirt* [3]).
3) A modification of the control register of a `virtio-balloon` emulated device, necessary for communication with respective driver at guest side. The *memory ballooning* mechanism is not used but it was just convenient to reuse existing virtual device and 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 `virtio-balloon` implementation.

Additionally, on the guest kernel side, we properly modified the `virtio-balloon` driver (`GM_drv`), correspondingly to the device part described above. Moreover, as a crucial component of the system, we developed the ARMv8 platform-specific part of memory *hot-plug* functionality in the *Linux* kernel, which had not been supported before. This part of the work has been published to respective community in the form of code patches [7].

B. Testbed configuration

The prototype implementation of presented architecture has been tested on the *Xilinx Ultrascale+ MPSoC ZCU 102 (rev. 1.0)* board, equipped with 4 *Cortex-A53* cores and hosting 4 GB of external memory [5]. The host OS is based on manufacturer's fork of *Linux* kernel, version *xilinx-v2016.4*, *QEMU* on upstream version *stable-2.5* and guest *Linux* kernel on upstream version *v4.14-rc8*.

By modifying the *device-tree* used by the host OS, we split the available RAM in two parts: 2 GB backing up the *local memory* and other 2 GB serving as an *isolated backends*. We configured the system so that guest memory section size is 512 MB.

C. Memory resize latency

Initially launched with 512 MB of RAM, a VM was scaled up to 2 GB in 3 steps (512 MB each), then scaled down in another 3 steps to return to the initial configuration. There were conducted 40 such rounds. Multiple samples from both *QEMU* and guest kernel side were collected and processed to obtain statistics presented in the table I. It characterizes delays of each resize stage, divided in two, referring to *scale-up* (upper half) and *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 2). This way the *total* latency is derived, that consists of *backend reservation* or *release* at host side plus the *request-reply* latency perceived by *QEMU*, that is the communication delay and respective reconfiguration at guest OS side (*guest subtotal*). The communication is based on accessing a device registers together with handling related interrupt and can be simply computed as:

	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 I: Latencies of memory resize steps

$$communication\_delay = request\_reply - guest\_subtotal$$

An order of the execution is reflected within both *Add* and *Remove* sections of table I. For the *scale-up*, memory *backends* are reserved before the guest can use it, and symmetrically for *scale-down*, the *QEMU* has to request the guest OS to relinquish memory sections first, before corresponding *backends* can be safely released. The signaling overhead is larger in the *scale-down* case because of additional interaction necessary at the beginning to trigger guest action first.

The ***guest subtotal*** values can be decomposed into three main steps, done symmetrically for *Add* and *Remove* part:

- Building or thrashing page tables used by the MMU (*memory management unit*),
- Initialization and cleaning of virtual memory map data structures,
- Pages onlining or offlining, that stands for passing or withdrawing pages to/from the memory allocator and flushing the TLB (*Translation Lookaside Buffer*) entries in the second case,

There are stages of a particularly high latency variance, deserving better comment.

First, regarding the page table building, it requires filling entries of 4-fold-nested address translation tables (default configuration in *Linux* kernel for ARMv8), which *tree-like* structure is determined by the MMU. Each entry of level-*N* table (that is one memory page) corresponds to one table of level-*(N+1)*. The level-1 table and some lower level ones 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 corresponding table is populated and additional allocations are more frequent. 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 case of *pages offlining* stage, when pages of a given section are reclaimed. They may

be owned by guest the memory allocator or entailed on *per-CPU* lists of each CPU core (in *Linux* a core is considered a separate CPU). Especially in the second case the operation may not be successful at the first attempt and is being repeated on each core until accomplished, otherwise the process cannot advance. As there can be multiple or none pages (of a given section) owned by per-CPU lists, the introduced delay may vary, depending on how long the system was running and what allocations and deallocations were performed by a given core.

The *section init* and *clean-up* stages embody the virtual memory map (so called *vmemmap*) management, that is `struct page` objects instantiation and clean-up, respectively. Although a delay of the *clean-up* step is also variable in the removal case, it does not contribute significantly to the overall latency.

In general, the execution time of 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 the *QEMU* and about 8% is consumed by communication between them. The biggest variability is contributed by the page table building stage. The median execution time of section removal equals to 320ms, with *QEMU* accounting for about 40% of it, guest for 50% and more than 11% devoted for communication. But the dispersion of this operation is much larger at the guest side as in the worst case it can take almost 5 times longer than the median value, mainly due to page offlining latency. Numbers in the table I are derived statistically from the set of samples for each stage so the above percentage values do not add up exactly to 100%.

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 occupied for the whole reconfiguration process and other cores could still execute the workload in parallel. The only difference is the *offlining* step, at which all cores have to release respective pages from their *per-CPU* lists.

V. Conclusions

Presented virtualization layer design enables memory balancing of *memory section* granularity and leverages on paravirtualized approach. The possibility of runtime memory adjustment depletes the need of restarting or migrating the workload to VMs of more RAM. Thus, such backup VMs, that would otherwise have to be kept ready for a quick workload handover, are no longer needed.

Thanks to isolation at the host level, the guest *memory sections* are physically contiguous, which makes the should ease deployment on disaggregated architectures. Together fixed size and predefined offsets of the sections, the logic of a dedicated host-level allocator may be simplified.

For an accurate RAM resize latency evaluation the most problematic step is page *offlining*. It can be affected by memory fragmentation level as well as the *per-CPU* lists gathering some pages. The former should be possibly limited, especially with regards to *memory sections* that would be first candidates to be hot-removed. The latter, instead, can be perhaps statistically upper-bounded for a given system configuration (*per-CPU* lists size) and a given workload (memory allocation patterns). Although the delay cannot be avoided, it would be at least predictable.

As a further work, *ballooning* could be adapted as a complementary fine-grained memory balancing, provided that it would be constrained to operate within one specific section only, in order not to render other sections non-removable. Such solution would be a trade-off between having limited page-based balancing and avoiding extensive guest memory fragmentation.

Acknowledgments

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.

References

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

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

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

[6] Kvm, linux kernel-based virtual machine, 2018. http://www.linux-kvm.org/.

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

[8] Qemu, system emulator and virtualizer, 2018. http://wiki.qemu.org/Main_Page.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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*, ANCS '14, pages 125–136, New York, NY, USA, 2014. ACM.

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

[18] 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.

[19] 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.