Official URL
DOI : https://doi.org/10.1145/3190508.3190537

Welcome to zombieland: Practical and Energy-efficient memory disaggregation in a datacenter

Vlad Nitu
IRIT, Toulouse University, CNRS, INP, Toulouse, France
vlad.nitu@enseeiht.fr

Boris Teabe
IRIT, Toulouse University, CNRS, INP, Toulouse, France
boris.teabedjomgwe@enseeiht.fr

Alain Tchana
IRIT, Toulouse University, CNRS, INP, Toulouse, France
alain.tchana@enseeiht.fr

Canturk Isci
IBM Research
canturk@us.ibm.com

Daniel Hagimont
IRIT, Toulouse University, CNRS, INP, Toulouse, France
daniel.hagimont@enseeiht.fr

ABSTRACT

In this paper, we propose an effortless way for disaggregating the CPU-memory couple, two of the most important resources in cloud computing. Instead of redesigning each resource board, the disaggregation is done at the power supply domain level. In other words, CPU and memory still share the same board, but their power supply domains are separated. Besides this disaggregation, we make the two following contributions: (1) the prototyping of a new ACPI sleep state (called *zombie* and noted *Sz*) which allows to suspend a server (thus save energy) while making its memory remotely accessible; and (2) the prototyping of a rack-level system software which allows the transparent utilization of the entire rack resources (avoiding resource waste). We experimentally evaluate the effectiveness of our solution and show that it can improve the energy efficiency of state-of-the-art consolidation techniques by up to 86%, with minimal additional complexity.

KEYWORDS

memory disaggregation, energy efficiency, virtualization

1 INTRODUCTION

In recent years, we have witnessed some tectonic shifts in the computing landscape. First, with virtualization and containerization technologies becoming mainstream, we were finally able to decouple applications and their operation environments from the underlying hardware. Then, with cloud computing democratizing access to compute infrastructures and platforms, we have transformed these into services that can be provisioned and consumed on demand. These advances not only changed how we design software and build systems today, but also opened up many new opportunities for improving computing efficiency and cost.

With virtualization came the simplified multitenancy of operating systems, consolidation, virtual machine (VM) migration, distributed, dynamic resource and power management [3]. These were aimed at improving the notoriously-low data center (DC) server utilization [4], reducing cost, and dramatically improving power efficiency. With the cloud, we were able to push the boundaries further. Economies of scale, advents of software-based availability enables us to keep compute devices simple, cheap and designed for perfect efficiency meeting observed demands. By continuously placing thousands, if not millions, of requests on these nodes we can keep them busy, highly-utilized, and working at their optimal point of energy efficiency. Essentially, with cloud and virtualization, we could consider the compute infrastructure as one giant computer that practically has infinite resources, yet operates nimbly, with almost perfect efficiency based on demand.

Unfortunately the reality has been far from this. After myriad projects, papers, products and services, we now have giant computers at our fingertips on demand that are fast, easy to use, yet still highly inefficient in their resource utilization and energy efficiency. The average compute node utilization in most cloud offerings is well below 50% [4–6]. So where has this gone wrong? One main reason behind the mismatch between our expectation and the reality is our inability to efficiently pack multidimensional application needs to the underlying bundled compute resources such as CPU, memory and network. And this is because what the infrastructure offered in its evolution did not meet what software demanded in its evolution. Over the last several years we have seen new applications emerge with vastly growing memory demands, while platform evolution continued to offer more CPU capacity growth than memory, referred to as *memory capacity wall* [12]. Therefore, we are unable to leverage consolidation, efficient packing and balanced utilization of resources in the cloud as memory demand direction saturates before the other dimensions.

This observation is actually one of the underpinnings of another significant shift that has been gaining momentum, namely *disaggregated computing* [12], which aims to change the server-centric

https://doi.org/10.1145/3190508.3190537

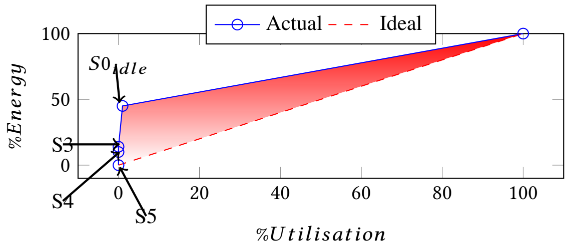

Figure 1: Energy consumption vs. server utilization. Solid line shows the common server power, while the dashed line plots energy-proportional behavior.

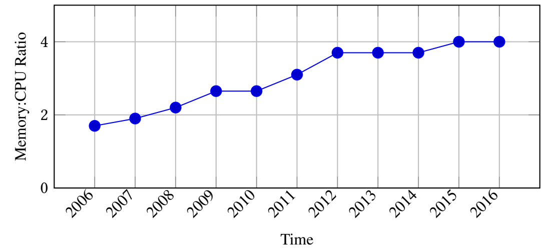

Figure 2: The $memory(GiB) : cpu(GHz)$ ratio for all introduced `m<n>.<size>` instances in AWS over the last ten years.

view of the infrastructure to a resource-centric view. In this model, each resource dimension can evolve and expand independently, and thus respond to evolving application demands. Disaggregated computing has the potential to lead us to our desired computing model that is nimble, boundless and highly resource and energy efficient. However, it is a solution for the long term that requires fundamental changes to compute hierarchy and operations.

In our work we explore a short-term solution that can have the benefits of disaggregation, yet that can be applied by introducing small changes to general-purpose computing hardware and virtualization/cloud software. Our solution targets the immediate problem at hand, disaggregating memory resources and unbundling them from other compute resources (e.g. CPU). We propose a new *Zombie (Sz)* ACPI state that is similar to suspend-to-RAM (S3) state in latency and power efficiency, but keeps the memory resources of a server active and usable by other nodes. In other words, a server in *Sz* state is a *Zombie* as it is brain-dead (CPU-dead), limps along consuming minimal resources (low-energy), but still has basic motor functions such as serving memory (memory-alive). We design a remote memory management protocol at the virtualization layer based on RDMA network interconnect that provides efficient access to the memory from a Zombie server, without requiring CPU intervention. Thus, all the servers that are in zombie state still contribute to the memory pool of the cloud, and yet have minimal additional energy footprint. As a result, we can pack VMs more densely in the cloud, achieving higher resource utilization and energy efficiency[1] with traditional hardware and virtualization software.

Even if the mainstream hardware does not currently support the *Sz* ACPI state, its implementation is fairly simple. *Sz* only requires completely independent power domains for CPU and memory. In order to evaluate the benefits of the Zombie technology, we developed *ZombieStack*, the software stack needed to leverage the *Sz* state. Even if we do not own an *Sz* compatible hardware, we estimated the energy consumption of a server in *Sz* state based on a model. The timing overheads related to remote memory are evaluated on ZombieStack by replacing the *Sz* servers with *S0* servers. Our experimental evaluations demonstrate that the *Zombie* technology improves energy efficiency on datacenter workloads by up to 67%, which is 86% better than state-of-the-art consolidation techniques.

[1]The low energy consumption of a Zombie server translates into less dissipated heat. Thereby, the Zombie technology also decreases the energy consumed by the datacenter cooling system.

In the rest of the paper, we first present some related background and motivation for our work. We introduce the *zombie (Sz)* state and its design in Section 3. We describe our RDMA-based remote memory management technique, and virtualization layer implementation in Section 4. We present *ZombieStack*, our OpenStack-based cloud implementation in Section 5. Then, we present our experimental evaluation in Section 6, highlighting the significant improvements with our approach. Last, we offer our conclusions.

2 MOTIVATION AND RELATED WORK

As we have discussed in the introduction, we have seen substantial opportunity and effort in improving resource utilization and energy efficiency with virtualization and cloud. There have been myriad efforts at the hardware, virtualization and the ensemble to attack this problem on multiple fronts, improving energy efficiency and overheads of low-power states and driving up server utilization and consolidation [3, 8–11, 58]. The motivation behind driving server utilization has been to improve consolidation ratios to reduce cost, while also benefiting from the widely-known observation that servers are more energy efficient (or *energy proportional*) at higher utilizations as depicted in Fig. 1.

While these prior techniques have improved utilization numbers significantly and improved energy-efficiency of systems, it is still difficult to actually reach server loads near 50% in even the most advanced implementations [4–6]. Lim et al. demonstrated that one main reason for this is a growing mismatch between platform resources and growing application demands [7, 12]. This is due to the combination of two opposite trends. First, we observe that emerging applications such as search, in-memory data stores, and analytics have developed a fast-growing appetite for memory resources to minimize request latencies, in response to real-time needs. This results in **a growing gap between memory and CPU demand** as memory demand has been growing much more rapidly than CPU demand. To validate this, we looked back at the historical instance sizes in AWS, and the observations were quite telling. As expected, AWS has gradually introduced newer-generation and bigger-size instances over time, as compute demands grew. However, when we look at the growth trend among different resources, we see that the memory configuration growth substantially outpaced that of compute. Figure 2 shows the ratio of memory size to CPU size for all AWS instances of family `m<n>.<size>`, where `n` is the generation and `size` the size attribute. The figure shows the general trend that while demand

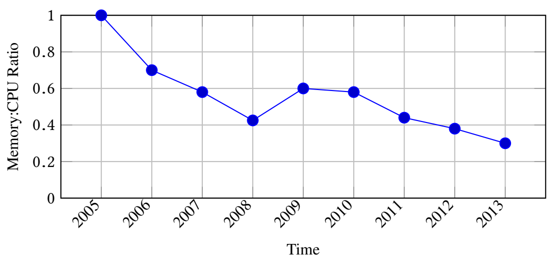

Figure 3: Normalized $memory : cpu$ **capacity ratio for multiple server generations.**

on both resources has grown substantially, the rate of growth for memory demand has been approximately $2X$ of CPU demand.

The second trend we observe is that **there is a growing gap between Memory and CPU supply in the reverse direction.** On the one hand, the International Technology Roadmap for Semiconductors (ITRS) estimates that the pin count at a socket level is likely to remain constant [15]. As a result, the number of channels per socket is expected to be near-constant. In addition, the rate of growth in DIMM density is starting to wane (2X every three years versus 2X every two years), and the DIMM count per channel is declining (e.g. two DIMMs per channel on DDR3 versus eight for DDR) [17]. On the other hand, another trend points the increased number of cores per socket, with some studies predicting a two-fold increase every two years [19]. If the trends continue, the memory capacity per core will drop by 30% every two years, as depicted in Fig. 3 [7].

These two opposing trends show that applications have been evolving in the direction where they require more memory than CPU, while servers are evolving to provide more CPU than memory. This situation leads to poor VM consolidation ratio [12, 23], thus energy waste as illustrated in Fig. 4(a). Several studies have investigated solutions to mitigate this issue. These can be grouped in three categories.

1. Reducing memory footprint at the server level: Several studies have tried to minimize VM memory footprint, thus increasing the consolidation ratio. These studies include page sharing [25, 26, 50], page compression [27–29], and ballooning [50–52]. There are three limitations of these solutions. They require non negligible computation (e.g., page sharing and page compression), they are intrusive (e.g., the balloon driver inside VMs), and they have limited returns with diverse workloads [51].

2. Resource sharing at the ensemble level: Leveraging the democratisation of high bandwidth low latency network adapters like Infiniband, some studies [18, 20–22, 24, 30, 31] tried to increase the memory utilisation by allowing servers to remotely use the residual memory of other servers. Although these solutions improve energy proportionality in comparison with the solutions presented above, the overall gain is too far from the optimal situation. The main problem is that one machine can remotely access the memory of another machine only through a remote active CPU even if that is not locally used. In addition, these solutions require substantial system modifications such as application-specific programming interfaces [32] and protocols [33], changes to the host operating system and device drivers [34, 35], and reduced reliability in the face of remote server crashes. Further, V. Anagnostopoulou et al. [65] define five different server power states along with a use case. However, the paper does not illustrate the system-level impact of a different power management on a virtualized cloud.

3. Resource disaggregation: Recent studies [12–14] claim that an emerging area for building energy-proportional data centers is to put an end to the server-centric (Fig. 4(a)) architecture, and move to a resource-centric architecture (Fig. 4(b)). This approach consists of decoupling all resources; each resource is assigned a dedicated motherboard and the ensemble is linked altogether to form a big computer. In this architecture, unused boards are powered-off. This is not possible in a server-centric architecture where a server's motherboard hosts all resources, thus powering it off will make all resources not accessible. Therefore, the resource-centric architecture leads to an optimal energy proportionality in comparison to the server-centric architecture (Fig. 4(a) vs. Fig. 4(b)). HPE [16] is one of the companies that massively invests in such a project. Nevertheless, it requires a total redesign of both hardware and software stack and thereby, it will take time until this solution becomes mainstream.

Micro-servers: An intermediate step proposed by manufacturers (e.g. HP Moonshot [36], Intel Rack-scale [38], AMD SeaMicro [39]) consists of: (1) first disaggregating network and storage devices from the $\{CPU, memory\}$ tuple, and (2) building *micro-servers* which include a limited number of $\{CPU, Memory\}$. Micro-servers are connected all together, sharing both the network and hard disk pools. The advantage of the micro-server solution comes from the fact that residual/unused resources are small since servers are small too (in comparison with commodity servers). However, this solution does not address the main issue, which is the disaggregation of the $\{CPU, memory\}$ tuple (recall that memory is the limited resource). The consequence is that a server's resource (e.g. memory) can be remotely used only if the server is powered-on. For instance, this is the case in AMD SeaMicro in which even the turn-it-off [43] feature cannot allow the remote utilization of a suspended micro-server's resource. This situation leads to a poor energy proportionality, as illustrated in Fig. 4(c).

Our solution: Memory disaggregation with *zombie* servers: In this paper we propose a new and less expensive way for disaggregating the $\{CPU, memory\}$ tuple. Our solution requires less hardware modifications than a full-fledged board level disaggregation. We rely on a simple approach to disaggregation at the power supply domain level. By this way, we allow a memory bank to be functional and remotely accessible via RDMA functions while a server is suspended. Such a server is called *zombie* and its corresponding ACPI state is noted as *Sz*. This disaggregation solution leads to much improved energy proportionality, which is not far from the ideal solution.

We present all the architectural trade-offs and energy efficiency characteristics of the discussed approaches in Fig. 4. Below we summarize the energy consumption characteristics of each approach[2] in units of *Emax*, the maximum energy consumed by a server at full utilization:

[2] The figures are rough approximations presented only for guidance.

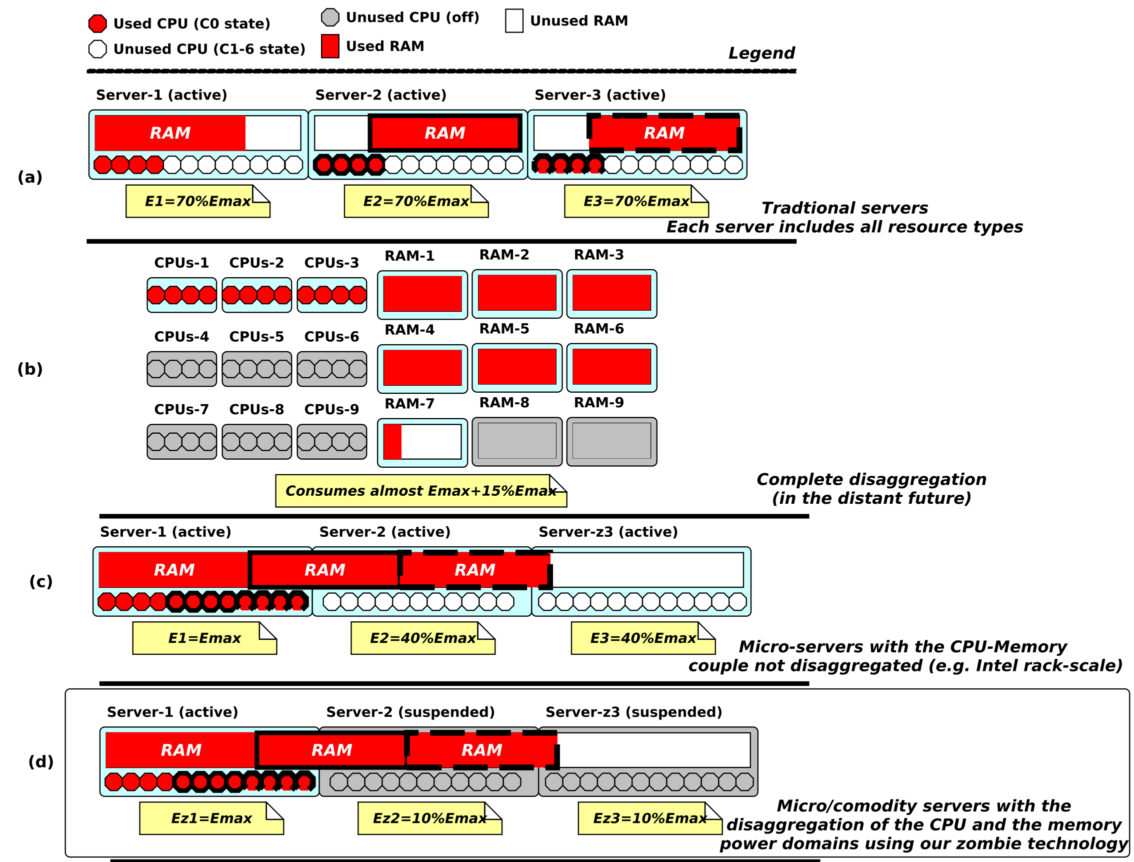

Figure 4: Resource disaggregation: summary of existing solutions. We illustrate each solution at the rack-level, considering a rack composed of three serves. We estimate the energy consumed by the rack in each solution. We can see that our proposition (d) results in the optimal energy proportionality while requiring less hardware and software modification.

- **Server-centric architecture (Fig. 4(a)):** $Total\ Energy\ Consumed = 2.1 \times Emax$.
- **Resource disaggregation, ideal case (Fig. 4(b)):** $Total\ Energy\ Consumed = 1.15 \times Emax$.
- **Micro-servers (Fig. 4(c)):** $Total\ Energy\ Consumed = 1.8 \times Emax$.
- **Our solution with zombie servers Fig. 4(d):** $Total\ Energy\ Consumed = 1.2 \times Emax$.

In summary, this paper makes four main contributions: (1) We introduce a new ACPI sleep state called the *zombie (Sz)* state (Section 3), (2) we describe and prototype a new practical rack-level memory disaggregation technique based on *zombie* servers (Section 4), (3) we present *ZombieStack* an OpenStack based cloud operating system that leverages our memory disaggregation solution (Section 5), and (4) we evaluate the timing overheads of *ZombieStack*, and we model and estimate the energy efficiency of the *Sz* state (Section 6).

3 *ZOMBIE* (*SZ*): A SLEEP STATE FOR SERVERS

The Advanced Configuration and Power Interface (ACPI) is a standard that allows an OS to perform power management on individual components (e.g. CPU cores, network adapters, storage devices, etc.) or the system as a whole. The global (system level) power states are named from *S*0 to *S*5. *S*0 represents the most active state (i.e. the CPU is running and executes instructions) while *S*5 is the most inactive one (i.e. the machine is turned off without saving any system state). *S*3 is an intermediate state also called Suspend-to-RAM. It cuts power to most of the components except the RAM memory, which stores the system state, the network adapter which is used to wake-on-LAN the machine and a part of the PCI/PCIe bus.

In this section we describe our new ACPI sleep state (S-state) called *zombie* or *Sz* state. The *Sz* state is similar to S3 state, with one key difference. It keeps its memory banks of the platform active and remotely accessible even when the server is suspended. Our main motivation in introducing this new *Sz* state is to address the growing gap between the memory demand vs. supply and the CPU demand vs. supply discussed earlier. With *Sz* state, an application running

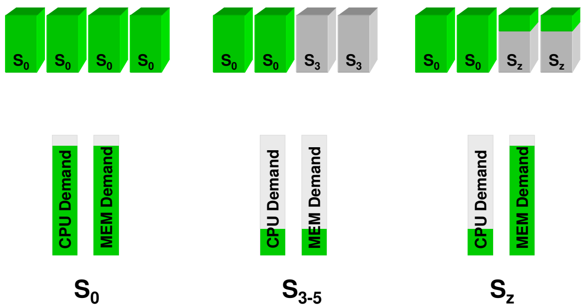

Figure 5: *Sz* state operation compared to S3 and S0.

on one platform can "borrow" memory from another, otherwise suspended, platform. This feature is provided neither by the ACPI specification nor by existing hardware or OS distributions.

The *Sz* state operates similarly to the S3 state for the most part. All components are turned off except the main memory and part of the network is kept active to serve remote memory requests. The memory behavior of *Sz* mimics that of *Si0x* state specifications, where the memory is kept in active idle, unlike the low-power self refresh mode of S3. The *Sz* State enables a nice compromise for a practical step towards disaggregated computing for memory. A general-purpose compute node can be used as a full-fledged platform when demand on resources is high, can be suspended to S3, S4 or S5 when demand is low, and can be kept in *Sz* state when compute demand is low, but the aggregate memory demand still requires the node to serve memory. Figure 5 shows the operation mode of *Sz* in comparison to the traditional S-states.

3.1 *Sz* State Design

The implementation of the new *Sz* state needs support from the manufacturer since it requires modifications across the stack from hardware and firmware to the OS, as well as to the ACPI specifications. At the hardware level, when a server enters ACPI S states, it follows a sequence to shutdown several power rails to the board components. As the memory and the networking logic for remote memory access need to remain active, power lines for these components require additional switches and control signaling for *Sz* enter/exit. State management hardware needs modifications to include the new S state and additional signals for triggering the right power state change actions for *Sz*. System management hardware needs additional signals from the participating chips for reporting and idempotence of actions. These signals are used to determine the state of the devices, when a state transition is active and to report the power state of the server. Firmware is involved in S-state transitions during boot up and during each *Sz* enter and exit. During boot up the firmware initialises *Sz* chipset configurations. During *Sz* enter and exit the firmware transitions individual devices to their corresponding S-states. The additional work required for the actual steps is minimal for *Sz* as most of the board is still transitioned to S3. Additional logic is required to transition memory and network to their active-idle states to enable their operation while the system is in *Sz* state. During *Sz* exit, once the chipset state is reinitialised, the firmware passes the control back to the OS to transition to general-purpose computation in S0 state.

```
1 +echo zom > /sys/power/state
2 +pm_suspend
3 enter_state
4 suspend_prepare
5 suspend_devices_and_enter
6 suspend_enter
7 acpi_suspend_enter
8 x86_ acpi_suspend_lowlevel
9 do_suspend_lowlevel
10 x86_acpi_enter_sleep_state
11 +acpi_hw_legacy_sleep
12 acpi_os_prepare_sleep
13 +tboot_sleep
```

Figure 6: The execution path to transition to the zombie state. It is similar to the S3 execution path, except the modifications on red functions (lines 1, 10 and 12).

We prototype the OS components of *Sz* state with the Linux Operating System Power Management (OSPM) framework. OSPM is the kernel component in charge of power management and shares this responsibility with the device-drivers. *Sz* state implementation in the kernel requires the modification of both the OSPM and the Infiniband device driver (*MLNX_OFED* in the prototype). This implementation starts from the *S3/S4* execution path, to which we applied slight modifications as presented in Fig. 6. We introduce a new keyword (*zom*) for triggering the transition to *Sz* when setting */sys/power/state*. We identify the set of devices which should be kept up during the *Sz* state (e.g., Infiniband card and its associated PCIe devices). The `pm_suspend()` call for these devices has been modified in order to prevent them from transitioning to the sleep state. The real activation of the transition is done by setting `PM1A` and `PM1B` ACPI registers. In the case of S3, `SLP_TYP` and `SLP_EN` are respectively set into these registers. Once set, `PM1A` and `PM1B` are read by the platform in order to know which state to transition to. Since this registers have unused values, we consider new ones for triggering to zombie.

4 MEMORY DISAGGREGATION USING *SZ* STATE

While the previous section focuses on the low-level ACPI adjustments which enable the *Sz* state, this section presents a practical approach to rack-level memory disaggregation which relies on the *Sz* state. Our technique leads to both high and balanced resource utilization and high energy efficiency.

In our rack-level management implementation servers are either *active* in *S0* state or *zombie* in *Sz* state. An active server can use its own memory or memory from other zombie servers. While our main contribution is on utilizing *Sz* state for energy-efficient memory disaggregation, our implementation also allows for serving and using remote memory from other active servers. If an active server requires more memory it can become a *user* of available remote memory. If there is capacity slack, workload is consolidated to fewer hosts to save energy, which then become zombies pushed into *Sz*. We implement two remote memory functions as (i) *RAM extension* (*RAM Ext* for short), and (ii) *Explicit swap device* (*Explicit SD* for short).

***RAM Ext*:** An ideal implementation of disaggregated memory as RAM extension would require special hardware interconnect for remote memory access, similar to NUMA [64]. Instead, we design a practical, simple solution based on commodity server and network architecture, and addressing the complexity in software. We implement a hypervisor-level swap mechanism, where the remote memory is presented as swap to the hypervisor. It keeps the frequently accessed pages in local memory and excess pages are simply swapped to the remote memory. One key advantage of our approach is that we simply build upon all the existing page promotion, relegation, hot page determination policies which are already built into the hypervisor. As a result, with a small set of tweaks and by leveraging hypervisor paging, we can transparently present remote memory to VMs running on the host. As demand decreases, pages are naturally swapped in, requiring no custom implementation for releasing remote memory.

***Explicit SD*:** As a natural extension of our remote memory design, a server may also use remote memory to implement swap devices to be provided to VMs. These memory-backed swap devices perform substantially faster than disk-based swap. Our implementation is similar to Infiniswap [62].

An interesting difference between these two remote memory functions is that, the VMs and applications are completely oblivious to the former function, which is hypervisor-managed, while the latter is fully-visible to those. Application behavior can be significantly different (particularly more aggressive regarding memory management) as it knows that fewer local pages are allocated to the VM (see the evaluation section).

4.1 Implementation

Fig. 7 presents our implementation architecture of a virtualized rack with the zombie technology. A general-purpose server in the rack plays one of the following five roles:

(1) **Global Memory Controller** (*global-mem-ctr*) manages the memory for the whole zombie pool. It is responsible for allocating/deallocating remote memory to servers.
(2) **Secondary Memory Controller** (*secondary-ctr*) enforces transparent high availability of the global controller. It monitors the main controller's state (periodic heart beat) and synchronously mirrors all operations.
(3) **User Server** (*server-A*) uses remote memory from other servers.
(4) **Zombie Server** (*server-C*) serves remote memory to other servers, while suspended in *Sz* state.
(5) **Active Server** (*server-B*) serves remote memory to other servers, while in active state.

All servers execute a **Remote Memory Manager** (*remote-mem-mgr*) agent, which interacts with the *global-mem-ctr* to request and release remote memory. The communication framework implements RPC over RDMA [24, 48]. In our implementation, the clients poll for the RPC results as RDMA inbound operations are cheaper than outbound operations. *Remote-mem-mgr* relies on low-level RDMA primitives instead of RPC calls to directly access remote memory and to implement *RAM Ext* and *Explicit SD* functions.

4.2 Initialisation

At startup, *global-mem-ctr* initialises various data structures for state keeping such as the list of zombie nodes. Initially all servers are designated active, and state is updated as they are pushed to *Sz*. Next *global-mem-ctr* starts a daemon serving the requests from *remote-mem-mgr* agents. Finally, it starts the mirroring and heartbeat processes for mirroring and high availability. *Secondary-ctr* spawns two processes to periodically monitor *global-mem-ctr* heartbeats and to establish the RPC over RDMA communication with the *global-mem-ctr* in order to receive the mirrored operations. Each *remote-mem-mgr* establishes an RPC over RDMA communication channel with the *global-mem-ctr* and initialises state to request and use remote memory.

4.3 Delegating and Reclaiming Server Memory

Here we first describe how servers can *delegate*, i.e., *lend*, their memory to *global-mem-ctr* via *remote-mem-mgr*. Then we explain how they can *reclaim* their memory when it is needed locally. As discussed previously, we have patched the OS of each server to implement the *Sz* state transition. When a server's OS receives the *suspend to Sz* signal, it signals its *remote-mem-mgr* to trigger memory delegation. *Remote-mem-mgr* computes free memory and organizes it in buffers. Their size (noted *BUFF_SIZE*) is uniform across the entire rack. It then notifies *global-mem-ctr* of its intention to go to *Sz* state via the `GS_goto_zombie(buffers)` function and communicates the list of zombie memory buffers it is lending via *buffers*. *Global-mem-ctr* uses an in-memory database to manage the allocation state of these buffers. Each remote buffer is characterized by an identifier, offset, size, its type (active/zombie), the host serving the buffer, and the server currently using this buffer (*nil* if it is not yet allocated to a server).

A zombie server can reclaim its memory once it becomes active again. Its *remote-mem-mgr* determines the amount of memory it wishes to reclaim (at buffer granularity) and informs the *global-mem-ctr* via `GS_reclaim(nbBuffers)`. *Global-mem-ctr* has to choose from its database which of the buffers belonging to this server will be returned. It first uses unallocated buffers and then chooses buffers allocated to other servers and reclaims them using the `US_reclaim(buff_IDs)` function. This function only informs the corresponding *remote-mem-mgrs* that `buff_IDs` are no longer available. As a result, the *remote-mem-mgrs* start transferring the backup copy of the data[3] to other remote locations. Last, *global-mem-ctr* returns the buffer identifiers to the reclaiming server. Once in possession of these buffers, the *remote-mem-mgr* of the server destroys the communication channels to these buffers and frees them.

4.4 Requesting and Allocating Remote Memory

Here we describe how a *user* server can request and allocate available remote memory from *global-mem-ctr* using the following functions:

`GS_alloc_ext(memSize)` requests a RAM Extension memory allocation of *memSize* that the *global-mem-ctr* must fulfill. This allocation is guaranteed by the cloud provider via admission control to

[3] Each write to a remote buffer (backing either a RAM Extension or an Explicit SD) is asynchronously mirrored to the local storage.

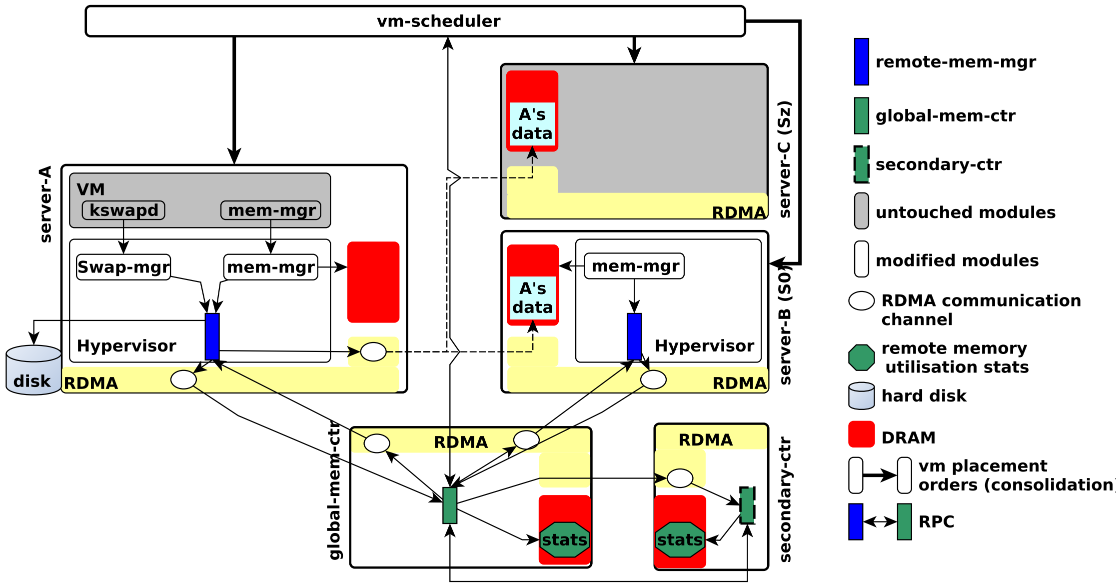

Figure 7: The architecture of a disaggregated rack provided by the zombie technology.

avoid rack-level memory overcommitment. Thereby, `GS_alloc_ext(memSize)` is called once at the VM creation time and returns a list of n_b buffers such that $n_b \times BUFF\_SIZE == memSize$

`GS_alloc_swap(memSize)` requests a VM Swap memory allocation of *memSize*. The full allocation is not guaranteed as it depends on the available memory in the rack. This allocation is best-effort because using a fast swap device is not included in the VM's SLA, contrary to RAM Extension. Therefore, this allocation is such that $n_b \times BUFF\_SIZE \leq memSize$. This function is periodically called (i.e. every 1 hour) in order to take advantage of unused remote buffers.

Memory from zombie servers have always higher priority than memory from active servers. Thereby, *global-mem-ctr* first attempts to allocate the requested memory from available free buffers. Next, it tries to get more remote memory from active and user servers with the `AS_get_free_mem()` and `US_reclaim(buff_IDs)` calls. For both, `GS_alloc_ext()` and `GS_alloc_swap()`, the *memSize* allocation is backed by memory from multiple remote servers. This approach minimizes the performance impact caused by a remote server failure. By default, all inactive servers are pushed into *Sz*. If the *global-mem-ctr* holds huge amounts of free memory (e.g. more than the total memory of a rack server), the cloud manager may decide to transition zombie servers to *S3* for further reducing the energy consumption.

4.5 Using Remote Memory

Here we describe how *user* servers use remote memory and our actual implementation for the KVM hypervisor [46]. As we previously discussed, user servers can utilize remote memory via two functions: (i) *RAM Ext*, and (ii) *Explicit SD*. Our *RAM Ext* implementation is a practical approximation to disaggregated memory, which operates transparently to VMs via our modified hypervisor-level swapping mechanism.

The ideal case of memory disaggregation requires fundamental changes to hypervisor memory virtualization implementation, where remote endpoints and page addresses need to be in shadow or extended page tables, and enabling direct access to these remote addresses. Such an implementation requires an important hardware evolution (TLB, MMU, DMA, cache coherency protocol, PCIe, etc.) [12]. In contrast, our solution relies on commodity, general-purpose servers[4], standard RDMA networking and a software-based solution with our modified KVM hypervisor, and unmodified VMs and applications.

Our modified virtualized memory management system within the hypervisor works as follows. Let *VMMemSize* be the amount of memory reserved by a VM. At VM startup, the hypervisor allocates a part of the server's local RAM (noted *LocalMemSize*) to the VM. If *LocalMemSize* is less than *VMMemSize*, the rest of the memory is provided by other remote servers as Extension memory. From VM perspective, all the memory is local and allocated in its pseudo-physical memory. From hypervisor perspective, the actual machine memory can be distributed between local physical and remote physical RAM.

We implement our solution in KVM's page fault handler, extending hypervisor paging to use remote physical memory buffers similar to swap devices. VMs are given pseudo-physical frames and the hypervisor manages their association with host-physical (machine) frames. KVM allocates physical frames on demand, which means when a VM modifies its guest page table and traps to the hypervisor, a physical frame is allocated and associated with the pseudo-physical page. In our solution, we provision both local and

[4]This servers are not yet for sale since they should implement our new *Sz* state as described in Section 3.1.

remote page frames to a VM. When a page fault is caused by a VM attempt to modify a guest page table, if a physical frame is available (free), the handler follows the traditional code execution path. Otherwise, it frees a physical frame to satisfy the page fault, using a page replacement policy.[5] Indeed, it asks the *remote-mem-mgr* for a remote page frame, transfers the content of the local frame to the remote frame, registers the information allowing its eventual reclaim and clears the present bit in the corresponding page table entry. When the page fault is caused by the non-presence of a page, we first check whether it is a page sent to a remote memory. If this is the case, a local page is allocated as above and the remote page is reloaded in the local page. Our paging policy keeps hot pages closer in local memory, and as local memory becomes scarce, demotes cold pages to remote buffers.

Our implementation of the second function, *Explicit SD*, is relatively simpler as no guarantee is offered to the VMs. Swap remote memory is obtained with the dedicated `GS_alloc_swap()` function. This function has the same prototype as `GS_alloc_ext()`, but the amount of returned memory may be less than requested as it depends on remote memory availability. Our *Explicit SD* implementation is based on the split-driver model [47]. When a VM is swapping-out a page to remote memory, the backend driver first contacts the *remote-mem-mgr* for allocating remote memory if available. It also asynchronously swaps to local storage for fault tolerance. When the *global-mem-ctr* reclaims this memory, the pages are still available on local storage and *remote-mem-mgr* uses this slower path to serve page requests.

5 CLOUD MANAGEMENT WITH *ZOMBIESTACK*

In the previous sections we presented the hardware implementation of *Sz* state (Section 3), and how we leverage *Sz* state for energy-efficient, practical memory disaggregation at the hypervisor level (Section 4). Here, we discuss the final layer of the compute stack, the cloud operating system. We describe how we leverage memory disaggregation with *zombie* servers for energy-efficient and practical cloud computing. We build a prototype cloud management platform, *ZombieStack*, based on OpenStack and our modified KVM hypervisor. We explain below the key cloud capabilities we introduce and the changes we did to the OpenStack components in our prototype.

5.1 Remote Memory Aware VM Placement

`Nova` is the OpenStack component responsible of VM placement on physical nodes. It operates in two phases. First, it filters the servers which are able to host the VM(s) and returns a list of suitable hosts. Second, it sorts these hosts based on certain placement criteria such as available resources and placement strategy (VM stacking or spreading). In our *ZombieStack* implementation we modify `Nova` to allow more relaxed filtering to account for remote memory availability. One trade off we explore in our implementation is the minimal amount of local memory needed for a host to be included in the list of suitable hosts. We answer this question with empirical evaluation. We perform several experiments using benchmarks with worst-case memory access patterns (see the evaluation section). Our results show that 50% local memory availability is a good, conservative compromise.

[5] We evaluated three policies (see Section 6.2)

5.2 VM Consolidation with *Zombie* Servers

Our VM consolidation implementation is based on OpenStack `Neat`. The consolidation algorithm employed by `Neat` can be outlined in four main steps [57]: Determine the underloaded hosts (all their VMs should be migrated and the hosts should be suspended); Determine the overloaded hosts (some of their VMs should be migrated in order to meet QoS requirements); Select VMs to migrate from overloaded hosts; Place the selected VMs to other hosts (wake up suspended hosts if necessary).

Vanilla `Neat` places a VM on a server only if the latter holds all the resources booked by the VM. In the same vein as VM placement, we modify this constraint to only check if 30% of the VM's working set size is available on the target server. If there is no host that satisfies this requirement, we choose and wake up a *zombie* host. We modified `Neat` so that it prefers zombie servers with the least amount of shared buffers. Neat calls `GS_get_lru_zombie()` which returns the *hostID* corresponding to the Zombie server having the minimum number of allocated zombie buffers. By this way, we minimize the amount of zombie memory which has to be reclaimed.

5.3 VM Migration Protocol

The vanilla pre-copy VM migration consists of only source and destination hosts that hold the VM's current and future memory state. As part of a VM's memory may be located remotely in our *zombie* implementation, the migration protocol of *ZombieStack* is more complex than traditional migration. In our implementation, the active part of VM memory is mostly local to the source server due to the replacement policy behavior. Any remote memory used for the VM consists of cold pages.

Our migration protocol implementation first creates a *listening VM* on the target host, similar to traditional migration. However, instead of iteratively pre-copying dirty VM memory pages, we follow an approach similar to post-copy migration [45]. We stop the VM and we copy its local active memory part (hot pages) to the destination host. The newly created VM can be resumed as soon as its active part is copied on the target host. An interesting side benefit of *zombie* servers is that the VM's remote memory needs no migration. Once started on the destination host, the active part can address its remote part in the same way as before. We just need to update the ownership pointers for the remote memory components. Overall, our disaggregated memory implementation with *zombie* servers somewhat complicates the orchestration of live migration. However, in addition to the energy savings benefits, disaggregation also improves migration performance by both reducing the migration overhead and by providing a natural decoupling of hot vs. cold VM pages.

6 EVALUATIONS

Our paper makes two main contributions which are: a new ACPI state (i.e. *Sz*) and a framework (i.e. ZombieStack) to exploit this board at the rack level. ZombieStack includes two utilisation modes namely *RAM Ext* and *Explicit SD*. Since the latter has been widely investigated in previous work [59–63], our evaluations focus on

RAM Ext while comparing it with *Explicit SD*. The second evaluation aspect we investigated is the energy gain that our solution can bring. Notice that each result presented in this paper is an average of ten executions. We do not present the standard deviation results because we observed stable results.

6.1 Experimental environment

Hardware. We used two environment types. First, we evaluated the effectiveness of ZombieStack using a real rack in our lab. This rack is composed of four HP compaq Elite 8300 machines (Intel Xeon Intel(R) Xeon (R) CPU i7, 16GB RAM, running Linux kernel 4.4) organized as follows: two machines for hosting the *global-mem-ctr* and the *secondary-ctr*, one machine services as a user server while the last machine plays the role of a zombie server. Having not yet *Sz* enabled boards, the zombie server is provided by an idle server in *S0*. The four servers are linked altogether with Mellanox Infiniband SB7800 switch. Each machine uses a Mellanox ConnectX-3 as the network card.
The second environment type is a simulator, used for the evaluation of ZombieStack in a large scale environment.
Software. We evaluated ZombieStack with both micro and macro benchmarks. The former is an application which iterates and performs read/write operations on the entries of an array whose size is configured at start time. Each entry represents a 4KB memory page. The performance metric of this benchmark is the execution time. Regarding the macro-benchmarks, we chose the following applications: Data Caching[6] from CloudSuite [40]; Elasticsearch nightly benchmarks [42][7]; and Spark SQL [37] with BigBench [44] (we used a 100GB data set and focused on query 23[8]). The performance metric of these benchmarks is the number of operations performed per second. Otherwise specified, every VM uses 8 processors.

6.2 RAM Ext's page replacement policy

The efficiency of *RAM Ext* depends on the replacement policy which selects the page that should be transferred to a remote memory when the local memory becomes scarce. We compared three common replacement policies:

- *FIFO*. The hypervisor records to a list (called FIFO list) the pages which generate page faults. The page to transfer is the one which has generated the oldest page fault.
- *Clock*. The hypervisor iterates through the FIFO list and chooses the first page whose "accessed" bit is zero. The "accessed" bit of all pages is periodically cleared.
- *Mixed*. The Clock policy is applied to the first x elements of the FIFO list (e.g. x=5). If no page is obtained, the FIFO policy is applied to the rest of the list. This policy is designed to reduce the costs associated with "accessed" bits' management and list iterations.

[6]Data Caching uses the Memcached data caching server, simulating the behavior of a Twitter caching server using a Twitter dataset.
[7]In respect to the page length, we only present the results for the NYC taxi benchmark. The NYC taxi data set contains the rides that have been performed in yellow taxis in New York in 2015. This benchmark evaluates the performance of Elasticsearch for structured data.
[8]BigBench includes more than 30 queries. We chose query 23 because it takes a lot of time to perform.

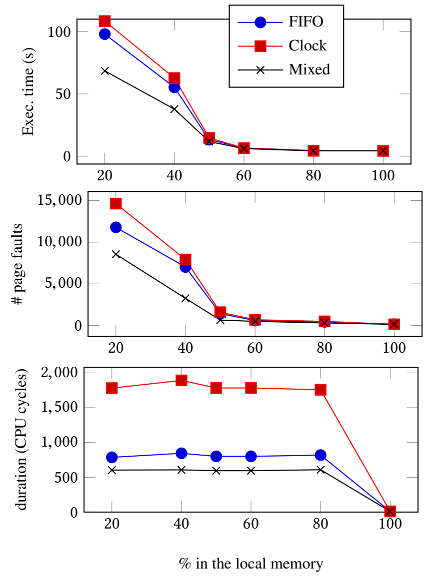

Figure 8: Comparison of three replacement policies (FIFO, Clock, and Mixed) for *RAM Ext*. (top) The micro-benchmark execution time, (middle) # page faults and (bottom) time taken by the policy to perform a page fault. Mixed is the best policy.

We relied on the micro-benchmark to evaluate the above policies. The benchmark runs inside a VM having 7GB reserved memory while its working set size (WSS) is configured to 6GB. The VM is launched on the user server. We performed several experiments while varying the proportion of its memory in that server. Its remaining memory is provided by the zombie server using *RAM Ext*. Fig. 8 presents the evaluation results. The collected data are: the execution time (top curve), the number of page faults caused by the replacement policy (middle curve), and the time taken by the replacement policy in the page fault handler (bottom curve). We can see that *Mixed* is the best replacement policy. This is explained by the fact that it minimizes the page list iteration time (which is fairly important, see the gaps in Fig. 8 bottom) while avoiding the replacement of a page which may be used in a near future (by checking the "accessed" bit, see the gaps in Fig. 8 middle). As a result, Mixed outperforms both FIFO (by up to 30%) and Clock (by up to 36%), see Fig. 8 top. Thereby, the remaining experiments rely on *Mixed*.

6.3 RAM Ext limitations

We investigated to what extent a portion of a VM's RAM can be provided by a remote server. To this end, we relied on both micro and macro-benchmarks. Recall that our micro-benchmark represents the worst-case application. The evaluation procedure for the macro-benchmarks is as follows. Given a benchmark, we first ran it with vanilla KVM in order to determine its maximum WSS that does not generate swap activities. This size will serve as the VM's reserved memory in *RAM Ext*. Afterwards, we ran the benchmark with ZombieStack-*RAM Ext* while varying the proportion of the VM's reserved memory in the local RAM. Table 1 presents the evaluation results in terms of performance penalty. We can see that

% in local mem	micro-bench.	Elastic search	Data caching	Spark SQL
20%	9k%	15.6%	9.6%	27%
40%	4k%	6%	3.16%	6.5%
50%	8%	4.2%	1.35%	5.34%
60%	2.1%	3.01%	0.35%	2.04%
80%	0.04%	0.01%	0.32%	0.2%

Table 1: Performance penalty evaluation when a proportion of the VM's reserved memory is provided by a remote server. 50% is a good compromise.

providing up to 50% of the VM's reserved memory with a remote RAM is a good compromise. It leads to an acceptable penalty, less than 8%. We can observe that this proportion is also appropriate for macro-benchmarks. Our results are consistent with the ones in [24]. ZombieStack is configured with 50% in order to take into account worse case applications like our micro-benchmark, even if such applications are rare.

6.4 RAM Ext compared with Explicit SD

Let us consider two VMs (noted v_1 and v_2) configured as follows. v_1's reserved memory is m and v_2's reserved memory is $m-x$, $x \leq m$. v_1 runs in ZombieStack-*RAM Ext* with $m-x$ memory provided by the local server. v_2 runs in ZombieStack-*Explicit SD* with a mounted swap device provided by the RAM of the zombie server. The size of this swap device is x. Let us consider that v_1 and v_2 run the same application. One may think that the performance of that application will be the same in the two VMs. To clarify the situation, we compared the two utilisation modes while extending the analysis to other swap device technologies including: a local fast swap device (provided by an SSD, Samsung MZ-7PD256), and a local slow swap device (provided by a HDD, Seagate ST12000NM0007). Table 2 presents the evaluation results in terms of performance penalty. The following observations can be made. (1) v_1 outperforms v_2, see Table 2 column 2-3. In fact, v_2 generates much more swap activities on the remote server than v_1. For instance, v_2 generates more than 122% traffic than v_1 in the case of Elastic search. This comes from the fact that most applications and operating systems are configured according to the RAM size they see at start time [51]. (2) Using a remote RAM as the swap space through Infiniband is better than using a local storage, even if the latter is fast (see Table 2 column 3-5). In addition, fast storages require additional costs, leading to an unacceptable performance per dollar for data center operators [53].

6.5 VM Migration

We compared our VM migration implementation with the vanilla live VM migration. To this end, we ran the micro-benchmark inside a VM with different WSS. We are interested in the time taken by the migration process. Fig. 9 presents the evaluation results. We can see that in the vanilla implementation, the migration time is almost not affected by the WSS. This is explained by the fact that the number of iteration performed by the hypervisor for transferring dirty pages is fixed; it does not depend on the memory activity. In ZombieStack, only the memory pages within the local memory (about 50% of the WSS - see Section 5) are transferred. Thus, our implementation outperforms the native one, especially when the WSS is low.

% in local mem	v_1-*RE*	v_2-*ESD*	v_2-*LFSD*	v_2-*LSSD*
Micro benchmark				
20%	9000%	49500%	∞	∞
40%	4000%	14500%	∞	∞
50%	8%	2300%	302000%	∞
60%	2.1%	3.02%	3400%	429000%
80%	0.04%	0.7%	2.01%	5%
Elastic Search				
20%	15.6%	43.2%	85.12%	∞
40%	6%	38.6%	68%	307%
50%	3.4%	17.1%	45.04%	105.8%
60%	0.2%	12.3%	17.4%	55.3%
80%	0.01%	0.8%	1.6%	3%
Data caching				
20%	9.6%	15.7%	140.8%	∞
40%	3.16%	6.4%	41.7%	∞
50%	1.35%	3.1%	18.2%	∞
60%	0.35%	1.1%	3.04%	∞
80%	0.32%	0.35%	0.68%	13.2%
Spark SQL				
20%	27%	31.64%	122%	∞
40%	6.5%	18.39%	63.23%	∞
50%	5.34%	13%	35%	∞
60%	2.04%	2.9%	11.45%	185.36%
80%	0.2%	0.3%	3.2%	4.78%

Table 2: The performance penalty (i.e. how much longer the execution takes?) depending on the local/remote memory ratio. *RAM Ext* (RE) vs *Explicit SD* (ESD) and other swap technologies (LFSD=Local fast swap device; LSSD=Local slow swap device).

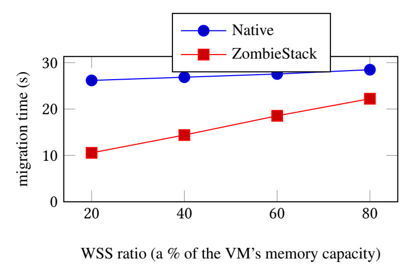

Figure 9: Comparison of the vanilla live VM migration solution with ZombieStack.

6.6 Energy consumption

6.6.1 Sz energy consumption.

Given that we don't have a HW prototype, we estimated the amount of energy that a machine would likely consume in the *Sz* state. To this end, we consider two machine types available in our lab: the one presented above (noted *HP*) and a Dell precision Tower 5810 (noted *Dell*). Using PowerSpy2, a power analyzer device, we measured the energy consumed by each machine in several configurations: *S0* without the Infiniband card (noted $S0WOIB$), *S0* with the Infiniband card not in use (noted $S0WIBOff$), *S0* with the Infiniband card in use (noted $S0WIBOn$), *S3* without the Infiniband card (noted $S3WOIB$), *S3* with the Infiniband card (noted $S3WIB$), *S4* without the Infiniband card (noted $S4WOIB$), and *S4* with the Infiniband card (noted $S4WIB$). Notice that a server in a sleep state usually keeps at least one of its network card (the Infiniband card here) in a power state which allows the Wake-on-LAN (WoL). This corresponds to $S3WIB$ or $S4WIB$. Table 3 presents the results. Knowing that *Sz* is a

	S0WOIB	S0WIBOff	S0WIBOn	S3WOIB	S3WIB	S4WOIB	S4WIB	*Sz*
HP	46.16%	52.20%	53.84%	4.23%	11.03%	0.19%	6.81%	12.67%
Dell	35.35%	42.33%	44.77%	1.97%	8.71%	1.12%	8.31%	11.15%

Table 3: Energy consumption of our two experimental machines in different configurations. Each value is the percentage of the machine's maximum energy.

kind of *S3* in which the RAM and the circuitry from the Infiniband card to the RAM are kept functioning, the energy consumed in *Sz* can be estimated as follows:

$$E(Sz) = (E(S0WIBOn) - E(S0WIBOff)) + (E(S3WIB) - E(S3WOIB)) + E(S3WOIB) \quad (1)$$

$(E(S0WIBOn) - E(S0WIBOff))$ is the energy induced by the Infiniband card activity; $(E(S3WIB) - E(S3WOIB))$ is the energy consumption which allows the WoL (i.e. the low-powered Infiniband card, PCIe, root complex, etc.). Using equation 1, we estimated the energy consumed by our testbed machines in *Sz* (see the last column of Table 3).

6.6.2 Energy gain in a large scale DC.

We evaluated the energy gain that can be achieved using ZombieStack in a DC. To this end, we relied on Google datacenter traces [56] which record the execution of thousands of jobs monitored during 29 days. Each job is composed of several tasks and every task runs within a container (seen as a VM is this paper). The total number of servers involved in these traces is 12583. The traces contain, among other information, for each task: its start time and termination time, its booked resource capacity (CPU and memory), its actual resource utilization level (gathered periodically). From these traces, we built a second set in which the memory demand is twice the CPU demand as the actual trends reveal (see the motivation section). Relying on these two set of traces, we simulated a DC which is equipped with the OpenStack consolidation system (i.e. Neat [57]).

We compared ZombieStack with Oasis [55], a consolidation approach oriented to energy-efficient cluster management. Oasis works as follows. After the execution of the consolidation plan, Oasis selects all underused servers (i.e. CPU utilization level lower than a threshold - 20% in this paper). Let us note *S* this set of underloaded servers. All *S*'s VMs which are idle (e.g. CPU utilization level lower than 1%) are partially migrated [58] to other servers. A partial VM migration consists in transferring only the working set of the VM. The remaining memory pages are relocated to a low power memory server so that the initial server can be suspended for energy saving. We assume that an Oasis memory server consumes about 40% of a regular server's total energy consumption, as stated in the original paper [55]. We performed experiments while considering that servers are either *HP* or *Dell* (see above). Fig. 10 presents the evaluation results. We can observe that ZombieStack outperforms Neat and Oasis. The best results are obtained with the modified traces (Fig. 10 bottom), where ZombieStack outperforms Neat and Oasis respectively by about 86% and 59% with *Dell* servers.

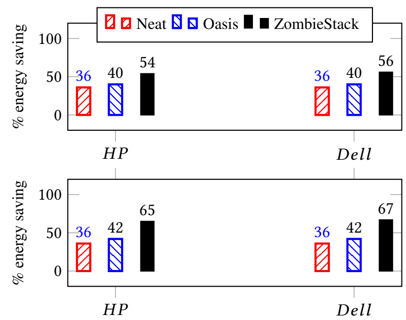

Figure 10: Energy saving: comparison with other resource management systems using both original (top) and modified (bottom) Google DC traces.

7 CONCLUSION

This paper presented a way requiring relatively little effort for disaggregating the $\{CPU, memory\}$ tuple based on the simple premise of making the power domains of CPU and memory independent. Assuming this change, we make the following contributions: (1) We described a new ACPI sleep state called *zombie* or *Sz* state. (2) We described a practical approach to rack-level memory disaggregation by leveraging *Sz* state. (3) We prototyped a cloud management platform, *ZombieStack*, based on OpenStack and a modified KVM hypervisor. We performed intensive experiments using macro-benchmarks and real DC traces (from Google clusters). We also compared our solution with existing ones (Neat and Oasis). The evaluation results showed that our solution is viable (acceptable performance degradation), leads to both high and balanced resource utilization and high energy efficiency.

8 ACKNOWLEDGEMENTS

We would like to thank Rodrigo Fonseca (our shepherd) and the anonymous reviewers for their helpful feedback. This work benefited from the support of Région Occitanie under the Prématuration-2017 program.

REFERENCES

[1] D. Patel and Amip J. Shah Cost Model for Planning, Development and Operation of a Data Center. HP technical report 2005, http://www.hpl.hp.com/techreports/2005/HPL-2005-107R1.pdf, accessed 2017-04/20.
[2] Urs Holzle and Luiz Andre Barroso The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. In *Morgan and Claypool Publishers*, 2009.
[3] VMware Inc. Resource Management with VMware DRS. Whitepaper 2006.
[4] Luiz Andre Barroso and Urs Holzle The Case for Energy-Proportional Computing. In IEEE Computer 2007.
[5] C. Delimitrou and C. Kozyrakis Quasar: resource-efficient and QoS-aware cluster management. In ASPLOS 2014.
[6] David Meisner, Brian T Gold, and Thomas F Wenisch The PowerNap Server Architecture. In ACM Transaction on Computer Systems 2011.
[7] Wm. A. Wulf and Sally A. McKee Hitting the Memory Wall: Implications of the Obvious. In SIGARCH Comput. Archit. News 1995.
[8] Canturk Isci, Suzanne McIntosh, Jeffrey Kephart, Rajarshi Das, James Hanson, Scott Piper, Robert Wolford, Thomas Brey, Robert Kantner, Allen Ng, James Norris, Abdoulaye Traore, Michael Frissora Agile, Efficient Virtualization Power Management with Low-latency Server Power States. In ISCA 2013.

[9] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin N. Vahdat, and Ronald P. Doyle Managing Energy and Server Resources in Hosting Centers. In SOSP 2001.

[10] Xiaoqiao Meng, Canturk Isci, Jeff Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis Efficient Resource Provisioning in Compute Clouds via VM Multiplexing. In ICAC 2010.

[11] Niraj Tolia, Zhikui Wang, Manish Marwah, Cullen Bash, Parthasarathy Ranganathan, and Xiaoyun Zhu Delivering Energy Proportionality with Non Energy-proportional Systems–Optimizing the Ensemble. In HotPower 2008.

[12] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, Thomas F. Wenisch Disaggregated memory for expansion and sharing in blade servers. In ISCA 2009.

[13] Krste Asanovic. Keynote. In FAST 2014, https://www.usenix.org/conference/fast14/technical-sessions/presentation/keynote.

[14] Huawei. High throughput computing data center architecture (HTC-DC) In http://www.huawei.com/ilink/en/download/HW_349607.

[15] ITRS Reports International Technology Roadmap for Semiconductors (SIA) 2007 Edition.

[16] HPE The Machine project. https://www.labs.hpe.com/the-machine.

[17] HPE Memory technology evolution: an overview of system memory technologies. In http://h20565.www2.hpe.com/hpsc/doc/public/display?sp4ts.oid=78193&docId=emr_na-c01552458&docLocale=en_US, accessed 2017-04/20.

[18] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In SIGMOD 2016.

[19] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick A view of the parallel computing landscape. In Commun. ACM 2009.

[20] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng HydraDB: a resilient RDMA-driven key-value middleware for in-memory cluster computing. In SC 2015.

[21] The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. In SoCC 2016.

[22] Rack-Scale In-Memory Join Processing using RDMA Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. In SIGMOD 2015.

[23] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker Network support for Resource Disaggregation. In HotNets 2013.

[24] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker Network requirements for resource disaggregation. In OSDI 2016.

[25] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman An Empirical Study of Memory Sharing in Virtual Machines. In USENIX ATC 2012.

[26] Grzegorz MiÅĆÃşs, Derek G. Murray, Steven Hand, and Michael A. Fetterman Satori: enlightened page sharing. In USENIX 2009.

[27] . Fred Douglis The compression cache: using online compression to extend physical memory. In Winter USENIX Conference 1993.

[28] Magnus Ekman and Per Stenstrom A Robust Main Memory Compression Scheme. In ISCA 2005.

[29] Jishen Zhao, Sheng Li, Jichuan Chang, John L. Byrne, Laura L. Ramirez, Kevin Lim, Yuan Xie, and Paolo Faraboschi Buri: Scaling Big-memory Computing with Hardware-based Memory Expansion. In ACM Trans. Archit. Code Optim 2015.

[30] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout It's Time for Low Latency. In HotOS 2013.

[31] Jinho Hwang, Ahsen J. Uppal, Timothy Wood, H. Howie Huang Mortar: filling the gaps in data center memory. In VEE 2014.

[32] Samir Koussih, Anurag Acharya, and Sanjeev Setia Dodo: A user-level system for exploiting idle memory in workstation clusters. In HPDC 1999.

[33] Michail D. Flouris and Evangelos P. Markatos The network RamDisk: Using remote memory on heterogeneous NOWs. In Cluster Computing 1999.

[34] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath Implementing global memory management in a workstation cluster. In SOSP 1995.

[35] Michael R. Hines Anemone: Adaptive Network Memory Engine. Thesis: A Florida State University Libraries, 2005.

[36] HPE Serveur Moonshot. https://www.hpe.com/fr/fr/servers/moonshot.html, accessed 2017-07/20.

[37] Apache Spark SQL. https://spark.apache.org/sql/, accessed 2017-07/20.

[38] Scott's Weblog Thinking About Intel Rack-Scale Architecture. http://blog.scottlowe.org/2014/09/22/thinking-about-intel-rack-scale-architecture/, accessed 2017-07/20.

[39] AMD AMD SeaMicro. http://www.seamicro.com/, accessed 2017-07/20.

[40] CloudSuite: A benchmark suite for cloud services. http://cloudsuite.ch/, accessed 2017-07/20.

[41] Yahoo! Cloud Serving Benchmark (YCSB). https://github.com/brianfrankcooper/YCSB/wiki, accessed 2017-07/20.

[42] Elasticsearch nightly benchmarks. https://elasticsearch-benchmarks.elastic.co, accessed 2017-07/20.

[43] AMD Data Sheet - SM15000. http://www.seamicro.com/node/254, accessed 2017-07/20.

[44] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen BigBench: towards an industry standard benchmark for big data analytics. In SIGMOD 2013.

[45] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan Post-Copy Live Migration of Virtual Machines. In VEE 2009.

[46] Kernel Virtual Machine. https://www.linux-kvm.org/page/Main_Page, accessed 2017-07/20.

[47] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon Software defining system devices with the 'Banana' double-split driver model. HotCloud 2014.

[48] Maomeng Su, Mingxing Zhang, and Kang Chen, Zhenyu Guo, and Yongwei W RFP: When RPC is Faster than Server-Bypass with RDMA. EuroSys 2017.

[49] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko StefanoviÄĞ, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In OOPSLA 2006.

[50] Waldspurger C A Memory resource management in vmware esx server. In OSDI 2002.

[51] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone Memory Application level ballooning for efficient server consolidation. In EuroSys 2013.

[52] Chiang J, Han-Lin Li, Tzi-cker Chiueh Memory Working Set-based Physical Memory Ballooning. In ICAC 2013.

[53] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron Migrating server storage to SSDs: analysis of tradeoffs. In EuroSys 2009.

[54] Vlad Nitu, Boris Teabe, Leon Fopa, Alain Tchana, and Daniel Hagimont StopGap: Elastic VMs to enhance server consolidation. In SPE 2017, DOI 10.1002/spe.2482.

[55] Junji Zhi, Nilton Bila, and Eyal de Lara Oasis: energy proportionality with hybrid server consolidation. In EuroSys 2016.

[56] Charles Reiss, John Wilkes, and Joseph L. Hellerstein Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, Nov. 2011. Revised 2012.03.20. Posted at https://github.com/google/cluster-data, accessed 2017-04/20.

[57] Anton Beloglazov and Rajkumar Buyya OpenStack Neat: A Framework for Dynamic and Energy-Efficient Consolidation of Virtual Machines in OpenStack Clouds. In CCPE 2014.

[58] Nilton Bila, Eyal de Lara, Kaustubh Joshi, H. AndrÃľs Lagar-Cavilla, Matti Hiltunen, and Mahadev Satyanarayanan Jettison: Efficient Idle Desktop Consolidation with Partial VM Migration. In EuroSys 2012.

[59] Maxime Lorrillere, Julien Sopena, SÃľbastien Monnet, Pierre Sens Puma: pooling unused memory in virtual machines for I/O intensive applications. In SYSTOR 2015.

[60] Nadav Amit, Dan Tsafrir, Assaf Schuster VSwapper: A Memory Swapper for Virtualized Environments. In ASPLOS 2014.

[61] Xi Li, Pengfei Zhang, Rui Chu, and Huaimin Wang Optimizing guest swapping using elastic and transparent memory provisioning on virtualization platform. In Frontiers of Computer Science: Selected Publications from Chinese Universities 2016.

[62] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang Shin Efficient Memory Disaggregation with Infiniswap. In NSDI 2017.

[63] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: it's time for a redesign. In Proc. VLDB Endow. 9, 7 (March 2016).

[64] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17). ACM, New York, NY, USA.

[65] Vlasia Anagnostopoulou, Susmit Biswas, Heba Saadeldeen, Alan Savage, Ricardo Bianchini, Tao Yang, Diana Franklin, and Frederic T. Chong Barely alive memory servers: Keeping data active in a low-power state. JETC 2012.