Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor

Vlad Nitu* Pierre Olivier† Alain Tchana* Daniel Chiba† Antonio Barbalace†
Daniel Hagimont* Binoy Ravindran†

*Toulouse University, †Virginia Tech

vlad.nitu@etu.enseeiht.fr, {Alain.Tchana, Daniel.Hagimont}@enseeiht.fr,
{polivier, danchiba, antoniob, binoy}@vt.edu

Abstract

The ability to quickly setup and tear down a virtual machine is critical for today's cloud elasticity, as well as in numerous other scenarios: guest migration/consolidation, event-driven invocation of micro-services, dynamically adaptive unikernel-based applications, micro-reboots for security or stability, etc.

In this paper, we focus on the process of setting up/freeing the hypervisor and host control layer data structures at boot/destruction time, showing that it does not scale in current virtualization solutions. In addition to the direct overhead of long VM setup/destruction times, we demonstrate by experimentation indirect costs on real world auto scaling systems. Focusing on the popular Xen hypervisor, we identify three critical issues hindering the scalability of the boot and destruction processes: serialized boot, unscalable interactions with the Xenstore at guest creation time, and remote NUMA memory scrubbing at destruction time. For each of these issues we present the design and implementation of a solution in the Xen infrastructure: parallel boot with fine-grained locking, caching of Xenstore data, and local NUMA scrubbing. We evaluate these solutions using micro-benchmarks, macro-benchmarks, and on real world traces. Results show that our work improves the current Xen implementation by a significant factor, for example macro-benchmarks indicate a speedup of more than 4X in loaded scenarios.

Keywords Virtualization, Cloud Computing, Elasticity, Scalability

1. Introduction

One of the key characteristics of today's cloud is that hardware and software resources are provided on demand, thereby rendering the resource pool a user can exploit *elastic* [9]. Elasticity provides benefits to customers, who can scale up and down their resource usage based on their clients' demands, while paying the provider in accordance with their need. At the same time, providers also benefit from elasticity as they can combine virtual resources of tenants with dynamic usage patterns and consolidate them on the same set of physical machines, while reducing, for example, energy consumption, and thereby costs. Elasticity requires VMs to continuously be booted, shut down, migrated, checkpointed and restarted – all operations that strongly depend on quick VM initialization and/or fast VM tear-down and resource liberation. In addition, swiftly loading Virtual Machines (VMs) in the cloud is also critical in various scenarios: event-driven invocation of micro-services, dynamically adaptive unikernel-based applications, and micro-reboots for security or stability, load balancing. etc. Unloading a VM is highly important as well, because it can for example block other VMs starving for resources on the same host.

This paper argues that the initialization and destruction of VMs in the cloud must be efficient, not only because it is becoming a central functionality, but also because these operations are pure overhead that do not hold any value for the provider and are usually not billed to the client [18, 39]. Moreover, it is common today to hear developers claiming long application deployment times in the cloud; initialization and destruction are contributing to such times [3, 35].

There are two types of initialization and destruction overheads that affect cloud elasticity. The first is direct overhead, which deterministically elongates the initialization and destruction times, and has been studied in previous work [17, 29, 30]. The second is indirect overhead, which manifests with variable waiting times in variable stages of the initialization and destruction processes. This paper targets both and demonstrates with experiments on a real world deploy-

ment using the Amazon EC2 that indirect overhead negatively impacts the efficiency of auto-scaling systems.

Various previous works focus on efficiently restoring the VM memory state for checkpoint/restart [21, 25, 43, 44, 46]. We focus the process of setting up/freeing the hypervisor and host control layer data structures at boot/destruction time, involved in both checkpoint/restart of stateful VMs as well as regular boot/halt of stateless, lightweight VMs such as unikernels. We show that this process does not scale in current virtualization solutions. In particular, we focus on the popular Xen [14] hypervisor for the case of paravirtualized guests, and we identify three critical issues hindering the scalability of the boot and destruction process: serialized boot forced by coarse-grained locking, non-scalable interactions with the Xenstore, and remote NUMA memory scrubbing overhead at destruction time. For each of these issues we present the design and implementation of a solution in the Xen infrastructure: parallel boot with fine-grained locking, caching of Xenstore data, and local NUMA scrubbing. We evaluate these solutions using micro-benchmarks, macro-benchmarks, and real world traces. Results show that our solution significantly improves the current Xen implementation: macro-benchmarks results show a speed-up of more than 1.5 times on a lightly loaded system, to up to more than 4 times when scaling the number of VMs in the system.

This paper makes the following contributions:

1. A demonstration of the indirect cost of VM creation/tear-down on cloud elasticity using real world deployment on EC2, motivating our work;
2. The identification of performance scalability bottlenecks in the VM creation/tear-down process of the Xen hypervisor;
3. The implementation of a new open-source version of the Xen hypervisor which makes the boot and shutdown of VMs faster, more scalable, and more stable;
4. A full evaluation of the proposed solution with micro- and macro-benchmarks as well as real-world traces.

Our version of Xen, as well as the sources of all experiments presented here, are available online (in an anonymous way): `https://github.com/xenboot/xenboot`.

Section 2 motivates this work by demonstrating the indirect cost of creation/destruction time on cloud elasticity. Section 3 gives a brief background on Xen. Section 4 describes the scalability issues of VM initialization in Xen and proposes a new design, while Section 5 presents a solution to improve VM tear down. In Section 6 the proposed solutions are evaluated. Section 7 overviews related work and Section 8 concludes.

2. Motivation

This paper focuses on VM initialization and destruction times, which may cause several cloud issues such as performance unpredictability as shown in this section. For the sake of brevity, we focus here on VM boot time only. Figure 1 shows the measurement of VM boot time in different situations: different hypervisors (KVM [20], VMware [13], Xen [14], and Amazon EC2 [5]) and varying the number of VMs booted in parallel (1 to 10). The EC2 experiments have been realized on the same *m3* dedicated host (not shared with other cloud tenants) using the *m3.medium* VM type. The server used in our lab for the experiments is equipped with 64 cores (Quad node Opteron 6376) and 128 GB of memory. Figure 1-top and Figure 1-middle respectively present for each experimental context the shortest and the longest measured boot time (e.g., in Xen they are given by $min/max(time\ xl\ create\ vm_1;time\ xl\ create\ vm_2;...)$). Figure 1-bottom presents the internal results of a specific experiment (booting 10 VMs on Xen). Note that the same behavior has been observed with other hypervisors. Several observations can be made. First, the shortest boot time is always the same regardless of the number of parallel boot processes, while the longest boot time increases linearly. Second, the hypervisor boots VMs sequentially, disregarding the number of available processors. Therefore, if n boot commands are sent to the hypervisor at the same time, $n-1$ commands are delayed. They are said to be penalized in the rest of the document.

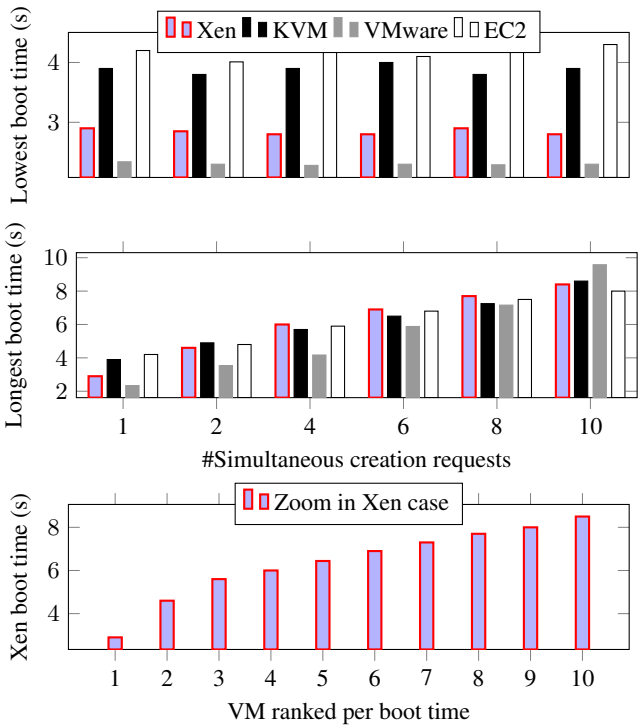

Figure 1. VM boot time measurement in Xen, KVM, VMware and EC2: (top) lowest values, (middle) longest values, and (bottom) a zoom in Xen for 10 parallel boot.

Takeaway *(C_1)*: **The boot time of a VM depends on the**

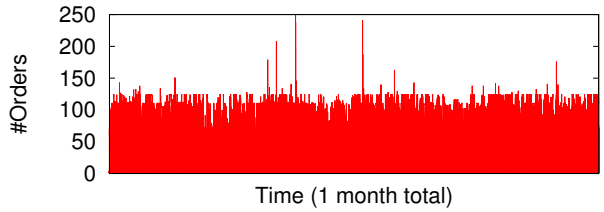

Figure 2. Number of simultaneous task creation orders computed from one month Google traces.

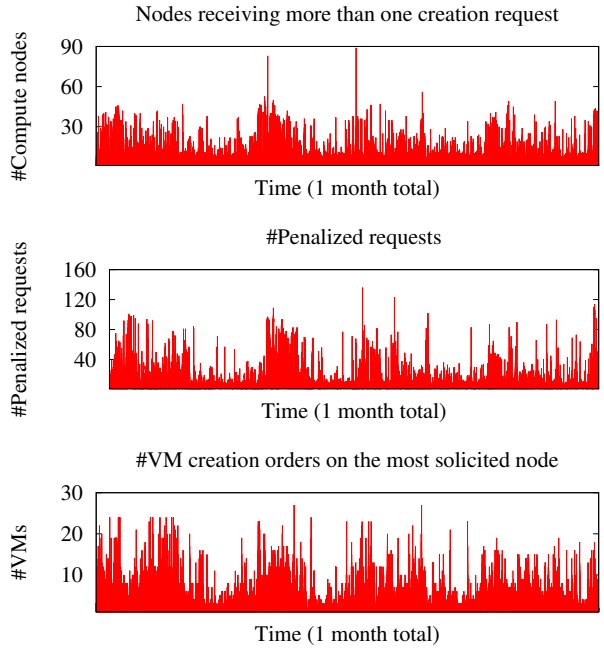

Figure 3. Congestion analysis on Google datacenter.

number of VMs that are actually booting, resulting in a potentially long and variable boot time. This situation is highly problematic in large-scale clouds (EC2, Microsoft Azure [35], Google cloud engine [16]) which receive so many VM creation requests that the same server can be solicited at the same moment for booting several VMs. The next paragraph demonstrates this situation by analyzing Google data-center traces [34, 41]. The description of these traces is presented in Section 6.

Figure 2 presents the number of task (and hence VM) creation commands handled by a Google data-center during one month. The high number of simultaneous commands received by the data-center scheduler increases the probability that the same *compute node* can be solicited to launch more than one VM at the same moment. Figure 3-top shows for each instant the number of compute nodes which received more than one VM creation request at the same time. The resulting total number of penalized requests is shown in Figure 3-middle. There are a significant number of penalized

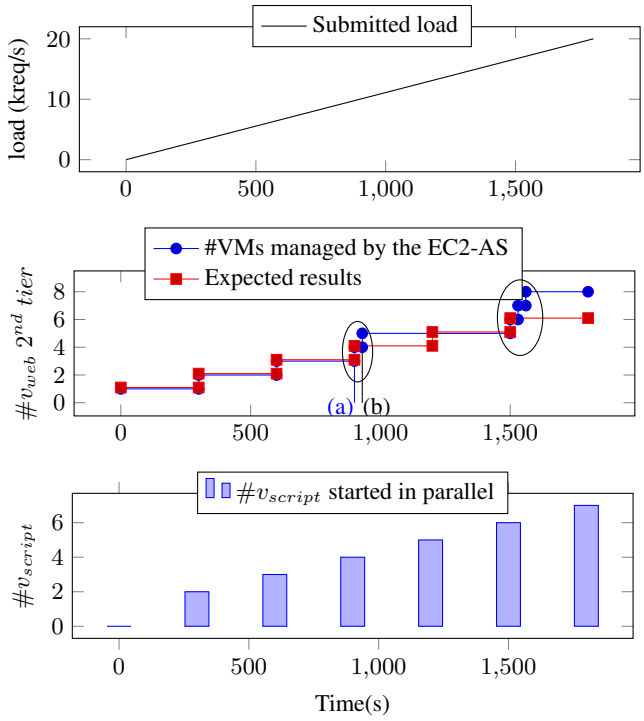

Figure 4. Variable boot time impact on EC2 auto-scaling, see encircled zones.

requests (up to more than 120 requests). Figure 3-bottom shows the number of requests (congestion) received by the most solicited compute node. The congestion level can be highly significant (up to more than 25 requests received by a compute node), resulting in a **longer** boot time. It is also variable, and knowing that VM creation commands initiated by the same user can be scheduled on any compute node, solicited with different intensity, a cloud user may observe a **variable** VM boot time.

Takeaway *(C_2)*: **According to the congestion on the solicited compute node, a cloud user may observe long and variable VM boot time for the same VM type.** A long boot time is problematic for applications which require quick scale-up (e.g., Netflix [37], unikernels [6]), while variable boot time could lead to unpredictable behavior. The next paragraph illustrates the consequence of the latter on the auto-scaling service of EC2 [2].

EC2, as well as other public clouds, offers the auto-scaling service (noted EC2-AS) [2]: the capability to automatically adjust the number of VMs of a user application in accordance with its workload. The implementation of this service follows the classical *observe-decide-act* feedback loop. To achieve this task, the auto-scaling engine relies on the user for configuring the feedback loop. The configuration of the *cool-down/warm-up period* is a delicate task because it directly affects the behavior of the EC2-AS. In fact, any further trigger-related scaling activity can only occur after this period. Indeed, its value should depend on VM boot

time, which is variable as shown above. Therefore, the question is: **how can the user statically configure this parameter (as imposed by EC2) knowing that VM boot time is variable?** A wrong configuration could lead to an unpredictable behavior as illustrated below. We run the following experiment on a single dedicated EC2 host. We deployed a 2-tier website receiving an increasingly demanding workload (see Figure 4-top). We denote v_{web} a VM used as a web application. The second tier of the web application is subject to the EC2-AS. The cool-down period is set to one second and the scale-up rate is one VM. We run in parallel a script which increasingly initiates parallel VM creation requests. We denote a VM created by the script as v_{script}. Figure 4 presents the results of this experiment. From Figure 4-middle we can observe several unexpected behaviors of the EC2-AS, denoted by encircled zones. We explain these anomalies by focusing on the first encircle zone. The EC2-AS observed a breach at time (a), resulting in the initiation of a VM creation request (denoted v_{web_a}). Surprisingly, another VM creation request (denoted v_{web_b}) has been initiated at time (b) whereas there is no burst between time (a) and time (b) (see the submitted load). This unexpected behavior happens because v_{web_a} boot time was much greater than the cool-down period, delaying the handling of the breach observed at time "a". This long boot time comes from the congestion that occurred at time (a) (see Figure 4-bottom). ***Takeaway (C_3):*** as reported by Netflix in [37] "*Auto scaling is a very powerful tool, but it can also be a double-edged sword. Without the proper configuration and testing it can do more harm than good.*" In the specific case, a wrong configuration can slow down VMs other than the target one.

2.1 Observations

In this section we motivated our work focusing on optimizing VM boot time. In this paper we are also interested in VM stopping time, which a potential source of issues in the cloud too, as it defines the time taken for freed resources to be available again. We believe these metrics are critical to the elasticity of the cloud. In summary, this paper tackles the following issues:

- Variable boot/stopping time: hinders predictability and indirectly impacts cloud elasticity, as shown above;
- Long boot/stopping time: directly impacts elasticity.

In addition to the impact on cloud elasticity, the VM setup and destruction times are critical in numerous scenarios as showed in past studies: checkpoint/restart of stateful VMs, or regular boot/halt for stateless, lightweight VMs such as unikernels. These must be as quick as possible when consolidating dynamic workloads [45], for example to save physical resources or energy consumption. It is also the case when using real-time/bidding-based pricing systems [11, 42] such as Amazon Spot Instances [1]. Moreover, VM creation/destruction must be swift in micro-reboot scenarios for fault tolerance [21] or security purposes [12].

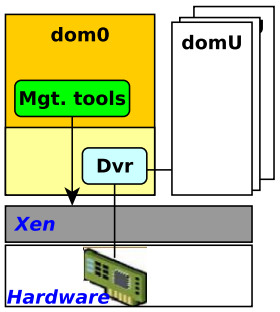

Figure 5. Xen architecture.

As previously mentioned, we focus on the process of setting up or destroying the hypervisor and control layer data structures needed for VM management, involved in both regular boot/halt and checkpoint/restart processes. Very few studies target this area, most of them focusing on optimizing the process of reinstating the machine memory state [21, 25, 43, 44, 46] for checkpoint/restart. We show in greater details in the next sections serious scalability issues in setting up VM management metadata in the hypervisor and control layer, taking Xen as a case study. We demonstrate that these issues are stronger in the case of small VMs (unikernels), for which their lightweight properties make that one wants to have a lot of them running on a single host. We also design and implement solutions to these scalability issues.

3. Background: Xen Hypervisor

3.1 General Overview

Xen [14] is a popular open-source Virtual Machine Monitor (VMM), or hypervisor, which is widely adopted by several cloud providers, such as Amazon EC2. It was initially designed to support para-virtualization [40], a virtualization technology in which the guest OS is slightly modified, also supported by several other hypervisors such as VMware. Xen is a type 1 hypervisor. In this model, the hypervisor is directly situated atop the hardware, taking the traditional place of the OS (see Figure 5). Thus, it has all the privileges and rights to access the entire hardware. It provides the capability of concurrently running several OSes in virtual machines (VM). In para-virtualization, VMs' OSes are modified to be aware of the fact that they are virtualized, reducing virtualization overheads. To this end, Xen provides a *hypercall* framework that is used by VMs when dealing with shared resources (e.g., new page table installation). Xen organizes VMs in two categories: the host OS (called *dom0*) and the others (called *domU*). Contrary to the *domU*, *dom0* is a privileged VM, responsible for hosting both device drivers and the Xen management toolstack (named *xl*). *dom0* can be seen as an extension of the hypervisor. This paper studies the

scalability of Xen management tools, specially those which are related to VM creation and destruction.

3.2 VM Initialization and Destruction

`xl create` is the command used in dom0 to initiate the creation of a VM. Its main execution steps are the following: (1) Checking the consistency of the VM configuration parameters (e.g. the existence of the VM file system). This step is performed at the dom0 level. (2) Creating the VM management data structures to make the VM manageable during its lifetime. This step is performed both at the dom0 and hypervisor levels. (3) Allocating resources to the VM (e.g. memory pages are granted to the VM). This step is performed at the hypervisor level. (4) Finally, executing the VM's kernel/-bootloader. The traditional boot process of an OS starts here. This step is performed at the VM level.

For its part, the destruction of a VM is triggered using either `xl shutdown/destroy` in dom0 or `halt` in the VM itself. It follows the inverse path of the initialization process: (1) VM's operating system shutdown, performed at the VM level. (2) VM resource freeing, performed at the hypervisor level. (3) Freeing VM management data structures, performed at both at the hypervisor and dom0 levels.

Intuitively, one can see that operations which are performed at the VM level are out of the cloud provider's purview. Indeed, they depend on both the VM kernel (built by its owner) and assigned resources (booked by its owner). This paper focuses on the steps performed either at the hypervisor or dom0 level.

4. Fast Parallel VM Setup

In Section 2 we saw that a VM boot time depends on the number of simultaneous VM boot requests. Here we deeply investigate and optimize VM boot process scalability.

4.1 Enabling Parallel Boot

4.1.1 Issue: Coarse-grained Locks

Figure 6 presents the key functions executed by the toolstack when creating a VM. First, the toolstack ensures that the hypervisor has enough available memory for accommodating the VM to be created, see `freemem` (line 6). Otherwise, the toolstack may ask dom0's balloon driver to deflate in order to deliver additional free pages to the hypervisor. The toolstack proceeds to the VM creation phase (`libxl_domain_create_new`) only if the hypervisor has enough available pages. However, between memory validation (`freemem`) and the actual consumption of memory pages (`xc_domain_claim_pages`) there is an interval which is subject to race conditions. Thereby, this section is protected by a huge lock (from `acquire_lock()` at line 8 to `release_lock()` at line 26). Since the lock have to be shared between *multiple VM creation processes*, the technical choice is a file lock (*flock*). This is the reason why VMs start sequentially regardless of the number of processors assigned to dom0. Figure 7 (the detailed version of Figure 1-bottom) shows that the time spent by the boot process waiting for the critical section increases linearly with the number of parallel boot processes. This is not the case for the other functions.

```
1 int main_create(int argc, char **argv)
2   create_domain(...)
3     ...
4     // parse the configuration file
5     libxl_read_file_contents(...);
6     ...
7     acquire_lock(); // acquire the file lock
8     ...
9     // ensure that there is enough free ↘
          memory
10    freemem(...);
11    |  // the VM's necessary memory
12    |  libxl_domain_need_memory(&need_memkb);
13    |   // hypercall to get free memory
14    |   libxl_get_free_memory(&free_memkb);
15    |   if (free_memkb >= need_memkb)
16    |     return true;
17    |   else ask dom0 to balloon
18    ...
19    // create the VM
20    libxl_domain_create_new(...);
21    |  ...
22    |  xc_domain_claim_pages(...);
23    |  ...
24    release_lock(); // end of critical section
25    ...
```

Figure 6. Key function calls during VM create.

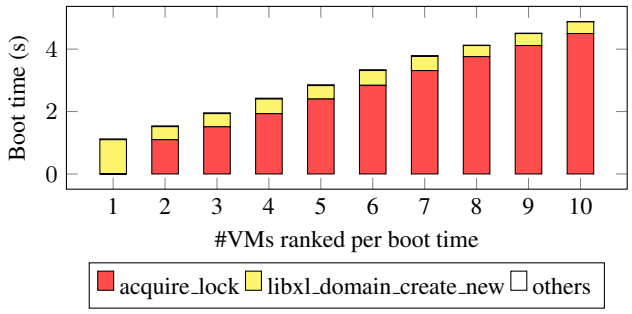

Figure 7. Figure 1-bottom presented in detail. The large majority of the actual VM creation code does not scale at all. (Notice that the value observed here are lower than those presented in Figure 1-bottom because we now focus only on operations performed at the hypervisor and dom0 levels.)

4.1.2 Solution: Fine-grained Locks

In a parallel boot, the vanilla Xen toolstack launches a process per booting VM. As presented above, this setup encounters a huge critical section. The basic idea behind our solution consists of introducing a new VM creation command (`xl client create <conf_file1> <conf_file2> ...`), which is able to start VMs in parallel. Figure 8 presents the algorithm of our new parallel create command in pseudo-

```
1 int free_memkb;
2 // master thread
3 int main_parallel_create(int[] need_memkb){
4   libxl_get_free_memory(&free_memkb);
5   for (i=0; i<argc; i++)
6     spawn_thread(&worker_task, &need_memkb[i↘
        ]);
7   // wait for worker threads
8 }
9 // parallelizable code
10 int worker_task(int need_memkb){
11   if (free_memkb >= need_memkb){
12     free_memkb -= need_memkb;
13   } else {
14     req_memkb = need_memkb - free_memkb;
15     balooned = baloon_dom0(req_memkb);
16     if (balooned){
17       free_memkb = 0;
18     } else
19       return ERROR;
20   }
21
22   created = libxl_domain_create_new(...);
23   if (!created){
24     free_memkb += need_memkb;
25     return ERROR;
26   }
27 }
```

Figure 8. Our parallel create algorithm.

code. First, we replace independent processes with *threads* sharing the same virtual address space. Subsequent to the command execution, the OS creates a main thread we call *master*. First, the master retrieves the available memory from the hypervisor and stores it in a shared variable (line 4). All write operations on this shared variable (line 12,17,24) are protected by a mutex. We have removed the locking/unlocking calls in the pseudo-code algorithm. Second, the master launches a *worker* thread for each VM to start (line 6). For simplicity, in Figure 8 we consider that the only important information for VM creation is the required amount of memory. Each worker thread first verifies that there is enough available memory for its VM (line 11). If so, it subtracts the needed amount from the shared variable (line 12) and proceeds to the VM creation (line 22). Otherwise, the worker thread asks dom0 to balloon and free some memory pages (line 15). If the VM creation process fails, the worker thread performs a rollback of the shared variable state. Notice that in our solution, the VM creation code (line 22) is outside of the critical section. In addition, we mention that our solution does not interfere with the legacy toolstack commands.

Optimization. Even if the scalability of our solution is clearly superior to the legacy creation toolstack (see Section 6.2.1), we still observed several limitations. First, the solution presented above needs another module at the cloud scheduler level (the component which handles cloud user requests). Indeed, the cloud scheduler should buffer for each compute node a set of VM creation requests (occurring within a configured time frame) in order to be able to initiate the parallel create command. Second, a group of requests have to wait for the complete execution of the previous group. Thus, if the buffered time frame is too small, we may fall into the same scalability problem as the legacy toolstack. Third, an important percentage of the creation process time is consumed in context validation and other operations that are orthogonal to the actual VM creation process. An example of such operation is the available memory verification (presented in Figure 8). The available memory can be retrieved after the hypervisor is booted, and stored in a shared variable. This variable could be maintained consistent with the real physical memory state and, therefore, we could avoid the cost of unnecessary hypercalls. That being said, the improved architectural step that we have implemented is to transform the *xl* toolstack into a daemon that listens for commands on a socket. During the daemon's design process, we have tried to identify all the elements which are independent of the actual VM creation. Thus, the context is prepared and warmed up when the daemon is launched. When a command is received on the socket, the daemon will spawn a worker thread to execute the command. We try to minimize the operation set executed by a worker thread. This architecture will further speed-up the VM creation process. The cut out of the huge file lock uncovered another scalability problem related to the Xenstore. The next section covers this aspect in more detail.

4.2 Scaling Interactions with the Xenstore

4.2.1 Issue: Xenstore Scalability when Increasing the Number of Domains

The Xenstore is a key-value store running as a daemon server in dom0. It is part of the Xen toolstack, and its goal is to store general information about the environment such as VM state, names, etc. It is extensively used during the VM create process to set up the hypervisor and dom0 data structures in preparation for the guest execution. We observed that the Xenstore becomes a serious bottleneck during VM creation when scaling the number of VMs running in the system. This is motivated by scenarios with hundreds of VMs running, as in the case with lightweight unikernel-based applications.

We measured the execution time of booting a unikernel based on Mini-OS [31] with 1 VCPU, executing an idle loop, while varying the number of unikernels running in the background. The background unikernels also execute an idle loop to avoid disturbance, and they are not scheduled on dom0 CPUs. To this end, we manually instrumented the Xenstore with a lightweight profiling infrastructure (as tools such as perf [38] and Oprofile [26] are inaccurate and limited with Xen [36]) to measure the time spent in the Xenstore. Profiling results are presented in Figure 9. On the top, one can notice the increase of `xl create` execution time with the number of running guests. Moreover, while for a few number of background VMs the time spent in the Xenstore is small (for example 23% for 2 VMs), it becomes

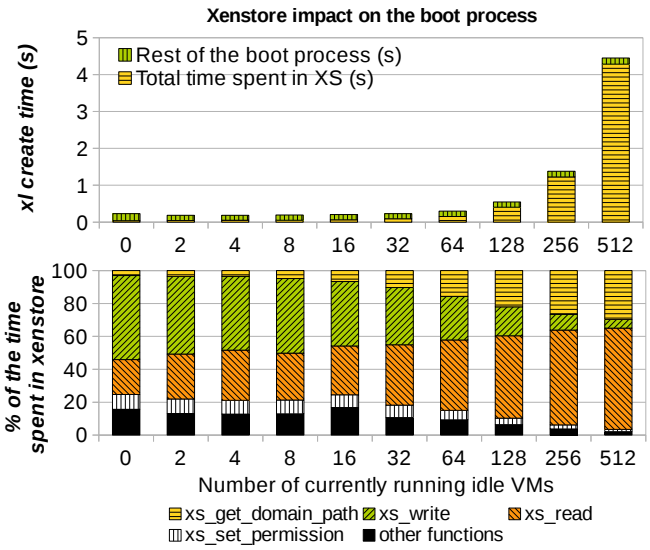

Figure 9. Time spent during VM creation in Xenstore (top) and per-function percentage breakdown (bottom).

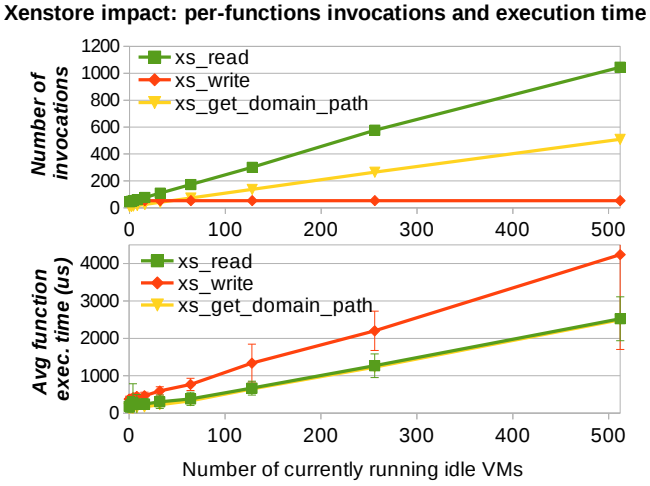

Figure 10. Evolution of the number of Xenstore calls (top) and average function execution time (bottom) while booting one VM according to the number of running VMs.

extremely large when increasing the number of background guests (96% with 512 guests). On the bottom of Figure 9 we present a per-Xenstore-function breakdown of that overhead. One can note that starting from 64 background VMs, more than 50% of the overhead is due to only two functions: `xs_read` and `xs_get_domain_path`. These two functions represent up to 91% of the overhead with 512 VMs, so our solution will focus on them. Overall, we found that booting one unikernel takes 232 ms when no other guests run in the background, 500 ms with 128 background guests, and up to 4 seconds with 512 background guests. Latencies over a second are unacceptable in many scenarios.

To analyze in detail the overhead due to the previously mentioned functions, Figure 10 presents the evolution of the number of calls and average execution time of each function while booting a single VM and varying the number of background guests. One can notice that both metrics increase linearly with the number of background VMs. In conclusion, the higher the number of VMs running in the system, the more Xenstore calls are made, and the longer they are. Our solution design is based on this observation. Note that there exists an alternative Xenstore version to the one we benchmarked (written in C), that is written in Ocaml [15]. We reproduced our experiments on this version and obtained a similar behavior: while the Ocaml Xenstore is slightly faster (7%) with few background VMs, it is actually slower than the C Xenstore when increasing that number (51% slower with 512 VMs).

4.2.2 Solution: Xenstore Data Caching

Tracking the cause of the overhead due to the Xenstore functions leads to two findings:

1. The increase in Xenstore calls' execution time with the number of running VMs is due to the following: with the default toolstack behavior, for each running VM there is a process running in Dom0 maintaining a connection to the Xenstore until the death of the domain. The performance of the Xenstore daemon decreases with the number of current connections due to the way the Xenstore server event loop is written;
2. The increase in terms of number of calls to `xs_read` and `xs_get_domain_path` is due to the fact that during the `xl` VM creation process, Xenstore calls are made in loops iterating over the currently running domains.

The solution to the first issue is simple - disabling the per-VM background process as we do in the parallel boot solution presented above (Section 4.1.2). This feature is already supported by Xen, although not enabled by default. Thus, we focus on providing a solution to the second issue: the increase in the number of calls to `xs_read` and `xs_get_domain_path`, made inside loops.

Analyzing the `xl` code related to the creation of a VM indicates that most of the calls to these functions are made into two *for* loops iterating over each currently running domain. One loop is calling `xs_read` to check the uniqueness of the created VM name. Another loop is calling `xs_read` and `xs_get_domain_path`, checking for each currently running domain whether it is a *stub domain* (a special type of VM used in Xen to offload Dom0 functionalities [24]). These findings explain why in Figure 10 the number of calls to `xs_read` is roughly 1000 and the number of calls to `xs_get_domain_path` is roughly 500 when there are 500 VMs running in the background.

To speed up the execution of `xl create`, we store the information required by the two loops in an in-RAM cache.

The cache simply consists of a tail queue where each element of the queue is a data structure containing the domain id, name, and a boolean value indicating whether the domain is a stub domain or not. We then replace the calls to `libxl_domid_to_name` and `libxl_is_stubdom` with calls to the cache look-up functions. The cache is populated lazily, so if the required information is not present in the cache, we make use of the default functions (`libxl_domid_to_name` and `libxl_is_stubdom`) to retrieve the information from the Xenstore, and then add it to the cache. We also ensure that the cache content is appropriately updated when a domain is renamed or destroyed.

5. Quick Domain Destruction

As mentioned in Section 2, we also investigate the VM destruction process.

5.1 Issue: Remote Scrubbing by Dom0 CPUs

Cache-coherent Non-Uniform Memory Access (cc-NUMA) enabled machines are composed of nodes, each node containing compute units (cores) and some memory. Cc-NUMA provides a unified view of the entire system to the OS / hypervisor, in particular concerning the memory. However, in NUMA systems there is a variable cost for memory access, depending on the distance between the core and the accessed memory. NUMA machines are supported by the Xen hypervisor and a lot of work has been done or is currently ongoing in the Xen community to solve the challenges related to NUMA [7, 8, 19, 33]. Key aspects include forwarding NUMA information to the guest, scheduling guest VCPU on the same NUMA node as the data they access, and allocating VM memory on the same NUMA node. We identified a scalability issue for Xen related to remote NUMA node accesses in the VM destruction process.

When a VM is destroyed, the hypervisor needs to fill its previously allocated memory with zeroes. This process, called *scrubbing*, is done for security reasons: without it, the fact that this memory may be later reallocated to another VM would result in information leakage. The issue we identified is the following: scrubbing is performed by the hypervisor upon receiving a domain destroy command (hypercall) from dom0. It is executed on the PCPU which is currently executing the dom0's VCPU from which the hypercall is originating. In the case where the memory of the VM being destroyed is present on a remote node, scrubbing is slow due to remote access. This case is very probable when considering large scale machines with a high number of NUMA nodes. To illustrate this issue, we ran several experiments with vanilla Xen on a machine with 64 cores and 128 GB of RAM, divided into 8 NUMA nodes of 16 GB each. We first validated the fact that destroy time can represent a significant bottleneck due to scrubbing. We used a minimalist Mini-OS based unikernel with one VCPU executing an idle loop, varying the VM allocated memory from 32 MB to 16 GB, and compared the boot and destroy execution times. We forced the memory to be allocated on the same NUMA node as dom0 to exclude any interference from the previously described issue. The results of the experiment show that when the amount of allocated memory is very small (less than 128 MB), the destruction process is faster than the creation: 3x, 2.1x and 1.4x for 32 MB, 64 MB and 128 MB respectively. However, when going above 128 MB the destruction time is significantly longer than the creation time: 2.13x, 9.5x and 16.2x for 512 MB, 4 GB and 1 GB respectively. We confirmed the fact that scrubbing was responsible for the long destruction time by disabling it and observing the time being reduced by 80% for a VM with 10 GB of memory. The destruction time itself can be significant, reaching 5.34 seconds for 16 GB. The performance loss combined with the significant variability in the destruction execution time are strong motivations to optimize that process.

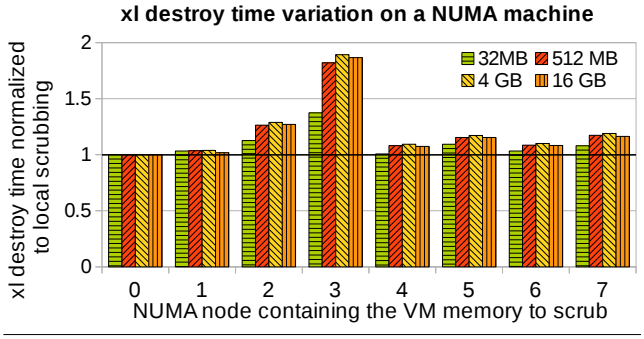

Figure 11. Remote scrubbing on NUMA machines.

To validate the remote NUMA scrubbing problem, we measured the destruction time of the previously presented Mini-OS based VM, while allocating 32 MB to 16 GB memory and forcing this memory to be allocated on a given NUMA node. This was done by pinning the unique VM VCPU to a PCPU belonging to the related NUMA node, knowing that Xen will allocate the memory on that node. The results of this experiment are presented in Figure 11, presenting the VM destruction execution times according to the NUMA node the VM memory is allocated on. The results are normalized to the performance obtained when allocating the VM memory on the same NUMA node as dom0 (node0). One can clearly see that the issue is confirmed, showing a increase in the execution time of around 10% for nodes 4 to 7, 20% for node 2, and up to 85% for node 3 (on that node, the destruction time represents more than 10 seconds).

5.2 Solution: Delegated Local Scrubbing

A naïve solution would be to give to dom0 one VCPU per NUMA node. However, this solution exhibits multiple drawbacks. First, this is not natively supported by Xen and would require a lot of re-engineering as currently Xen only supports giving n pinned VCPUs to dom0 starting from PCPU 0 up to PCPU n. Second, spreading dom0 VCPUs on each NUMA node would probably lead to a strong performance

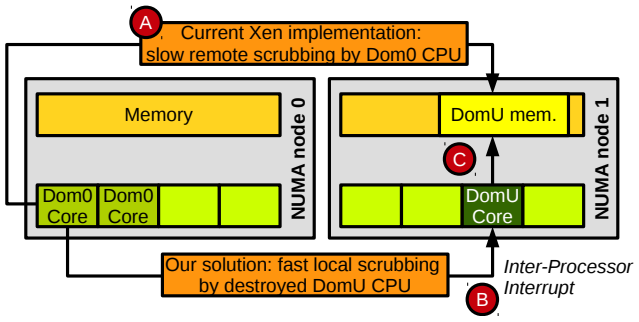

Figure 12. Remote scrubbing by the default Xen implementation (A), and our local scrubbing solution (B, C).

degradation due to the more frequent NUMA remote memory accesses for dom0's own memory, which would lead to a global performance loss for the entire virtualized environment due to the critical nature of dom0.

We chose to modify the scrubbing process performed in the hypervisor. A high level view of our approach is illustrated in Figure 12: the regular Xen implementation is depicted by point (A) while our solution is marked by point (B). In our solution, the dom0 CPU receiving the destruction order sends an Inter-Process-Interrupt (IPI) to one of the CPUs on the relevant NUMA node (B), that performs the actual scrubbing. More precisely, we deliberately choose one of the CPUs used by the destructed domU in order not to disturb the execution of the other VMs. In that way, we delegate the scrubbing process to a CPU local to the related NUMA node. Concerning the implementation, we removed at destruction time the calls to the scrubbing function, `scrub_one_page`, and add a pointer to the structure representing the page to scrub (`struct page_info`) to a linked list. The scrubbing function is called by `free_domheap_pages`, still executed by the dom0 CPU in our solution. When the list size reaches a certain value, the IPI is sent to one of the CPUs of the destructed VM. In the IPI handler, that CPU actually scrubs the pages and removes them from the list. In the meantime, the dom0 CPU spins on a condition that is the list being empty, and returns back to executing the destruction process when the scrubbing CPU is done. One can configure the threshold size of the list leading to the scrubbing process invocation. Obviously, triggering it each time there is a page to scrub hurts the performance because of the IPI overhead. We settled on a threshold value of 128 pages as we did not notice any improvement while increasing this value further. At the end of the VM destruction process, any page left in the list is scrubbed synchronously with the destroy process before it finishes. We deliberately chose to perform the scrubbing synchronously with the destruction process. Having it happen in the background, for example using idle CPU cycles, would speed up the destruction process itself. However as the amount of spare CPU cycles is difficult to estimate, this would lead to a lot of uncertainty about the time at which the memory will actually be available for other VMs to use [27].

6. Performance Evaluation

In this section we describe the performance evaluation process we adopted to demonstrate the benefits brought by our solution over a regular Xen implementation. We first realized a per-feature evaluation (Section 6.2) and an evaluation of the full system containing all contributions (Section 6.3). Theses evaluations are based on ad-hoc micro-benchmarks. Last but not least a large scale evaluation of the full system is presented in Section 6.4. This evaluation relies on both a reference macro-benchmark and Google datacenter traces.

6.1 Hardware and Software Platform

We used a server-class machine containing a quad AMD Opteron Processor 6376 CPU with $16 * 4 = 64$ cores. The server is equipped with 128 GB of RAM. The platform is composed of 8 NUMA nodes, each containing 8 cores (half a chip) and 16 GB of RAM. On the software side, we implemented our solution in Xen 4.7, more precisely the tag `RELEASE-4.7.0` on the Xen official git repository. dom0 is set up with 4 VCPUS pinned on PCPUs 0-3, and is configured with 4 GB of memory. Guest VMs use PCPUs 4 to 63. Unless otherwise specified, each VM runs a unikernel based on Mini-OS executing an idle loop. It is configured with a single VCPU and 32 MB of memory (the minimum amount of memory needed by Mini-OS).

6.2 Per-Feature evaluation

6.2.1 Parallel Boot

For this experiment we have created VMs executing a virtualized Ubuntu 16.04LTS. Fig. 13 presents the duration when VMs are created using (1) regular Xen, (2) our parallel solution and (3) our optimized daemon. First, we may observe that even for a single VM, our daemon is capable of creating a VM faster than the legacy toolstack. As we explained in section 4, this is due to the fact that most of the initialization and context verification are already done when the first creation request arrives. Second, we notice that our solution shows a better scalability even for reduced order parallel requests. This is down to the fact that we have replaced the huge file lock with fine grained locks. These improvements lead to an increased parallelization.

6.2.2 Xenstore Data Caching

To evaluate the Xenstore cache, we measured the execution time to create a single VM in a system with a variable number of currently running VMs. We also measured the impact of disabling the background process (called daemon in this section) attached to each running VM because it impacts the Xenstore scalability (as explained in section 4.2), and because it allows us to isolate the performance gains due to the cache itself. Moreover, we profiled the reduction in number

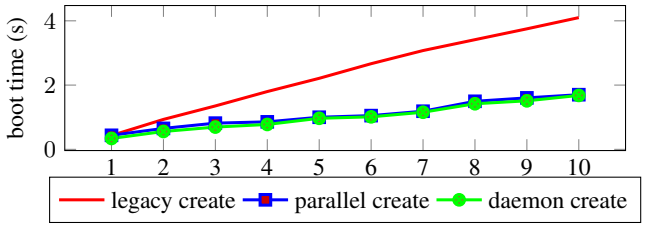

Figure 13. Boot time comparison between the legacy create, our parallel create and the optimized daemon.

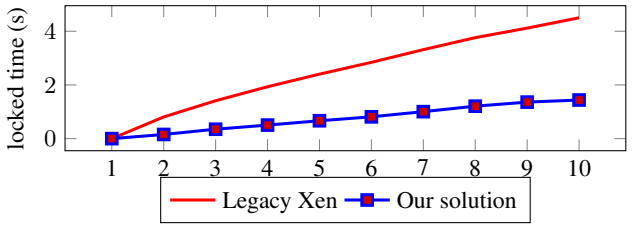

Figure 14. Parallel boot of 10 VMs. The figure presents the total time spent by its boot process in the critical section. Comparison between the legacy Xen and our solution.

of calls to `xs_read` and `xs_get_domain_path` thanks to the cache. The evaluation results are presented in Figure 15. At the top, the execution time of creating one VM with the regular `xl` implementation is compared to the two optimizations: daemon disabled without the Xenstore cache, and daemon disabled with the cache. At the bottom we present the percentage of time saved by enabling each optimization compared to no optimization. One can observe that for low number of running VMs (under 64), the cache does not help much as the number of calls to the Xenstore stays low, while disabling the daemon yields a 10% to 20% performance gain. With a high number of running VMs, the impact of the cache is significant. For example, in addition to the gains brought by disabling the daemon, the cache reduces the latency by 32% / 46% for 256 / 512 VMs running. This yields very fast create time even in the presence of numerous running VMs, for example 0.23s with 256 guests, and 0.30s with 512 guests. We further investigated the number of calls made to the Xenstore and confirmed that the cache makes it constant, irrespective of the number of running VMs: when creating one VM, 8 calls to `xs_get_domain_path` and 12 calls to `xs_read` are made, in a total of 117 calls to the Xenstore. This is a 93% reduction compared to the regular Xen implementation.

We also observed that the cache brought significant savings in VM destruction time, as the Xenstore is also involved in that process. We noted 33% of reduction in VM destruction time with 128 or 256 VMs running, and 47% of reduction with 512 VMs running, in addition to the gains brought by disabling the daemon.

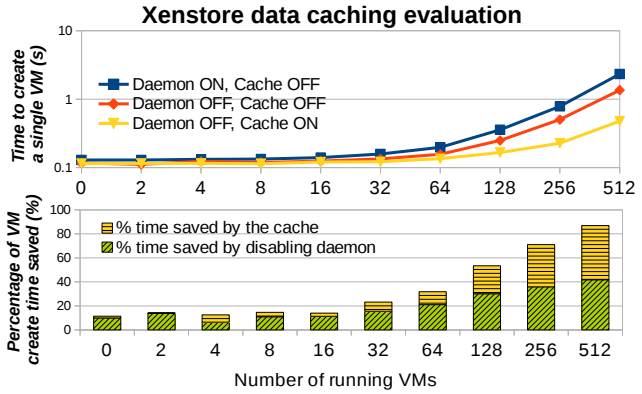

Figure 15. Xenstore cache impact on VM creation time (top) and percentage of create time saved by the cache / disabling `xl` daemon (bottom)

6.2.3 Delegated Local NUMA Scrubbing

To evaluate the benefits brought by the delegated NUMA scrubbing implementation, we measured the execution time of `xl destroy` for one VM and varied the NUMA node containing its memory. We also varied the memory allocated to the VM from 32 MB to 16 GB (size of a NUMA node). The results of this experiment are presented in Table 1. It presents the VM destruction time for each point in the experiment space normalized to the time for destroying the VM with the regular Xen scrubbing implementation on NUMA node 0, which is local to dom0 CPUs (seen as the ideal case). **X** is the regular scrubbing implementation, and **D** is our delegated scrubbing solution. One can see that without the delegated scrubbing, the VM destruction time is significantly longer when executed on a remote NUMA node (node 1-7), on average by 21%. This is especially true on node 3 where the performance loss can be up to 91%. This value is also highly variable: the standard deviation of the performance loss being 24%, leading to non deterministic VM destruction times. Concerning the delegated scrubbing solution, the performance loss is negligible: it takes a maximum value of 3% on node 0 due to the IPI overhead. When considering the entire set of results, the average overhead compared to the ideal case is 2.4%, and the standard deviation is 0.8%. A more detailed view of the results for a VM with 16 GB of RAM is presented in Figure 16, where one can observe the performance gains on remote nodes as well as the stability of the solution. These results show that our delegated scrubbing solution brings benefits in terms of (1) reduced and (2) stable and deterministic execution time for Xen's VMs destruction process.

6.3 Putting it all Together

To evaluate our system with all the optimizations (parallel boot, Xenstore cache, NUMA scrubbing) switched on, we measured the total execution time of booting and destroying 512 VMs, comparing our system with a vanilla Xen setup.

			Guest allocated RAM									
			32M	64M	128M	256M	512M	1G	2G	4G	8G	16G
NUMA node ID (0 = local to Dom0 CPUs)	0	X	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		D	**1.01**	**1.03**	**1.02**	**1.01**	**1.03**	**1.02**	**1.02**	**1.03**	**1.02**	**1.02**
	1	X	1.01	1.04	1.03	1.03	1.04	1.03	1.03	1.03	1.03	1.01
		D	**1.01**	**1.03**	**1.02**	**1.02**	**1.02**	**1.03**	**1.02**	**1.03**	**1.03**	**1.00**
	2	X	1.13	1.18	1.21	1.24	1.27	1.28	1.9	1.30	1.30	1.27
		D	**1.02**	**1.02**	**1.03**	**1.03**	**1.02**	**1.02**	**1.02**	**1.02**	**1.02**	**1.01**
	3	X	1.38	1.52	1.65	1.75	1.83	1.86	1.89	1.90	1.91	1.87
		D	**1.03**	**1.02**	**1.03**	**1.02**	**1.02**	**1.02**	**1.03**	**1.02**	**1.03**	**1.01**
	4	X	1.05	1.05	1.07	1.08	1.09	1.09	1.09	1.10	1.10	1.08
		D	**1.01**	**1.02**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.01**
	5	X	1.08	1.11	1.12	1.14	1.15	1.16	1.17	1.18	1.18	1.16
		D	**1.03**	**1.02**	**1.02**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.01**
	6	X	1.05	1.08	1.09	1.08	1.09	1.09	1.10	1.10	1.10	1.08
		D	**1.01**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.01**
	7	X	1.10	1.14	1.14	1.16	1.17	1.18	1.18	1.20	1.20	1.16
		D	**1.01**	**1.03**	**1.02**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.03**	**1.06**

Table 1. Domain destruction time normalized to the ideal, local scrubbing case. **X** is the Xen regular scrubbing implementation, and **D** is our delegated scrubbing solution.

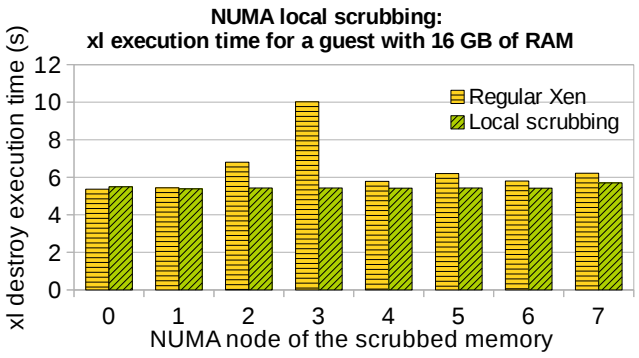

Figure 16. `xl destroy` execution time of a VM with 16 GB of memory, with the regular scrubbing implementation and our optimization

Regular Xen took 13 minutes 57 seconds to boot the VMs, and 1 minute 45 seconds for the destruction. Our system booted the VMs in 1 minute 51 seconds, and destroyed them all in 42 seconds, which is an 87% and 40% performance improvement for creation and destruction, respectively.

6.4 Real World Benchmarks

6.4.1 NPB Benchmark

We built a Mini-OS guest implementing the serial *Integer Sort* benchmark from the NAS Parallel Benchmark (NPB) suite [10]. We used the classes A and B, representing respectively small and medium data sets, in order to represent relatively short-lived, computation intensive jobs: a single execution of each class in a unikernel takes 5.08s (class A) and 17.57s (class B) in a regular Xen implementation, boot and destruction time included. This choice of short-lived jobs is

NPB IS Class	A			B		
Number of background ilde VMs	0	128	256	0	128	256
Vanilla Xen Jobs per minute	87.8	81.2	24.4	23.8	19.6	22.0
Optimized Xen Jobs per minute	**136**	**128.8**	**117.8**	**31.2**	**29.6**	**29.8**

Table 2. Jobs per minute for NPB classes A and B

deliberate, as it allows stressing the boot/destruction process. For each class, we create, as fast as possible, VMs running the benchmark during a five minute period, check the number of successfully finished benchmarks at the end of this period, and compute the number of jobs (i.e. executions of one benchmark) per minute. We compare our system to a regular Xen implementation. Each experiment is also relaunched with 128 and 256 idle unikernels running in the background. VMs are pinned in a round robin way on PCPUs. Note that because of the short runtime of the benchmark, our 64 cores platform is never saturated (at any given time there are no more than 64 active unikernels). The results of this experiment are presented in Table 2. Concerning class A, with no VM present in the background, our system yields a 54% performance improvement, mostly due to the parallel boot with fine-grained locking. When increasing the number of background VMs, the creation process is accelerated by up to 4.8x with 256 background VMs thanks to the Xenstore cache. While regular Xen takes an important performance hit with 256 background VMs, a small loss can also be observed on our system (14%), which is due to the `xl` daemon being activated (see Section 4.2.2) as it provides a practical way to automatically destroy a guest when the benchmark returns. Note that the comparison is still fair as the daemon is activated for both vanilla Xen and our system.

For class B, our system yields a 31% performance improvement. The performance loss observed from switching to this class is related to the amount of memory needed for the VM (512 vs 256 MB for A), imposing a longer boot time for both our system and vanilla Xen. As a consequence, the number of VM creations in a 5 minute time frame is considerably smaller, lowering the gains brought from the parallel boot. About the Xenstore cache, one can observe few differences from increasing the number of background VMs. We then conclude that the number of VM creation is too few in the case of class B to benefit from the cache.

6.4.2 Google Traces

We have also evaluated our solution with Google datacenter traces [34] analyzed in Section 2. They represent the execution of thousands of jobs monitored during 29 days. Each job is composed of several tasks and every task runs within a container. The total number of compute nodes involved in these traces is 12583. The traces contain, among other information, for each task the amount of resources allocated to its container, its hosted machine, its start time and termination time. Using a Hadoop cluster we were able to extract curves

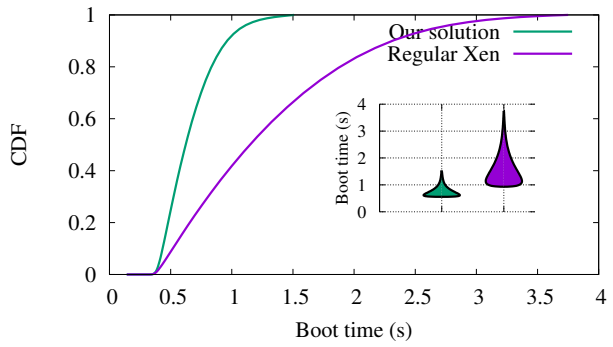

Figure 17. Comparison of our solution with vanilla Xen on Google datacenter traces presented in Figure 3-left-bottom.

presented in this section and also Section 2. This section evaluates the benefits of our system over the regular Xen implementation using these traces. To this end, we focus on all situations where a compute node of the datacenter receives more than one task creation order at the same time (see the two leftmost curves in Figure 3). We evaluated the boot time of each VM resulting in these situations. This evaluation has been performed by simulation as follows: relying on micro-benchmark results, we built two accurate boot time estimation functions representing respectively the regular Xen implementation and our solution. Figure 17 shows both the cumulative distribution function (CDF) and the violin representation of the boot time of each evaluated system. We can see that with our solution, almost all VMs boot within one second while it is the inverse in regular Xen. Further, the maximum boot time in regular Xen is about 4x of the maximum boot time obtained with our solution. The small gap we can observe between the minimum and the maximum boot times in our solution demonstrates its stability, important for predictability. This is not the case in the regular Xen implementation.

7. Related Work

As VM creation time is a critical metric in the elasticity of cloud-based systems, a fair amount of work has been done in studying and optimizing that process.

Several studies perform an analysis of VM startup (and destruction) time [17, 29, 30]. The numbers presented in these papers are consistent with our observation made in Section 2 and Section 4: in particular, concerning a VM boot time, the authors note that (1) the average as well as the variance increases with the number of concurrent VM boot requests and (2) requesting to boot an additional instance in an environment with already running VMs is significantly more costly than the average cost of an initial deployment in an empty environment. These studies focus on VM boot/destruction based on very high level parameters such as the time of day, instance type, VM image size, etc. The observations are also high level, providing few explanations for the observed results. We believe our detailed analysis presented in Section 4 can shed some light on the behavior observed in these papers.

Multiple papers focus on optimizing the VM creation time, for both regular [21, 22, 25, 43, 44, 46] and lightweight VMs [28, 32, 45]. With regular and stateful VMs, fast instantiation is achieved by cloning an existing VM or loading a previously made VM snapshot, using a checkpoint-restart system. In these cases, studies focus on optimizing the bottleneck that is reinstating the VM memory state, which can take tens of seconds even for small working sets [23]. This can be achieved by lazy/on-demand loading mechanisms [21, 25] combined with memory access predictions [43, 46] or heuristics-based pre-fetching [44]. We believe our work is complementary to these studies, and can bring further gains as we focus on other sources of overhead.

Concerning lightweight VMs, while [45] also relies on checkpoint-restart (for containers through CRIU [4]), in [28] the authors present a work that is relatively similar to ours, optimizing Xen VM creation time for unikernels. Optimizations include parallel device attachment, as well as Xenstore transactions parallelization. Once again, we believe our work is complementary: while in [28] an optimized Xenstore model is developed, we do not modify this software but reduce its usage through a cache. Moreover, contrary to our work, this paper as well as most of the work optimizing VM creation time do not consider situations where multiple VMs must be setup in a small time frame, or scenarios involving significant amounts of VMs running on a single host.

Finally, related to our scrubbing optimization, a patch [27] was proposed on Xen development mailing list allowing to delay the scrubbing process and perform it during idle CPU cycles. Contrary to our solution in which scrubbing is done synchronously with VM destruction, [27] has the drawback of letting a completely arbitrary amount of time be spent before the memory is available again, introducing a lot of non-determinism.

8. Conclusion

Long VM creation/tear-down times directly hinder cloud elasticity. In this paper, we also demonstrate that non-deterministic VM creation/tear-down overheads have an indirect impact by disturbing auto-scaling systems. Focusing on the popular Xen hypervisor, we identified three issues impeding the scalability of the current VM creation/destruction process: serialized boot forced by coarse-grained locking, non-scalable interactions with the Xenstore, and remote NUMA memory scrubbing. For each of these issues we designed and implemented a solution within the Xen hypervisor and toolstack: parallel boot with fine-grained locking, Xenstore data caching, and local NUMA scrubbing. Eval-

uation using micro and macro-benchmarks as well as real world traces show a substantial performance improvement over vanilla Xen, up to 4x for macro-benchmarks in a loaded system.

The enhanced Xen version source code, as well as the scripts for all experiments presented in this paper, are made available online (in an anonymous way to satisfy the double blind review process) here: `https://github.com/xenboot/xenboot`.

References

[1] Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/, accessed 2016-11-30.

[2] AWS Auto Scaling. https://aws.amazon.com/autoscaling/, accessed 2016-11-30.

[3] Why does azure deployment take so long? http://stackoverflow.com/questions/5080445/why-does-azure-deployment-take-so-long, accessed 2016-11-30.

[4] CRIU Wiki. https://criu.org/Main_Page, accessed 2016-11-30.

[5] Amazon Web Services. https://aws.amazon.com/, accessed 2016-11-30.

[6] Unikernel.org website. http://unikernel.org/, accessed 2016-11-30.

[7] Xen on NUMA Machines. https://wiki.xen.org/wiki/Xen_on_NUMA_Machines, accessed 2016-11-30.

[8] Xen 4.3 NUMA Aware Scheduling. https://wiki.xen.org/wiki/Xen_4.3_NUMA_Aware_Scheduling, accessed 2016-11-30, 2016.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721672. http://doi.acm.org/10.1145/1721654.1721672.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[11] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. N. Tantawi, and C. Krintz. See spot run: Using spot instances for mapreduce workflows. *HotCloud*, 10:7–7, 2010.

[12] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 189–202. ACM, 2011.

[13] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[14] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[15] T. Gazagnaire and V. Hanquez. Oxenstored: an efficient hierarchical and transactional database using functional programming with reference cell comparisons. In *ACM Sigplan Notices*, volume 44, pages 203–214. ACM, 2009.

[16] Google. Build What's Next Better software. Faster. https://cloud.google.com/, accessed 2016-11-30.

[17] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey. Early observations on the performance of windows azure. *Scientific Programming*, 19(2-3):121–132, 2011.

[18] T. Hoff. Are long vm instance spin-up times in the cloud costing you money? http://highscalability.com/blog/2011/3/17/are-long-vm-instance-spin-up-times-in-the-cloud-costing-you.html, accessed 2016-11-30.

[19] K. Z. Ibrahim, S. Hofmeyr, and C. Iancu. Characterizing the performance of parallel applications on multi-socket virtual machines. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2011.

[20] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, Ontario, Canada, June 2007.

[21] T. Knauth and C. Fetzer. Dreamserver: Truly on-demand cloud services. In *Proceedings of International Conference on Systems and Storage*, pages 1–11. ACM, 2014.

[22] T. Knauth, P. Kiruvale, M. Hiltunen, and C. Fetzer. Sloth: Sdn-enabled activity-based virtual machine deployment. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 205–206. ACM, 2014.

[23] I. Krsul, A. Ganguly, J. Zhang, J. A. Fortes, and R. J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 7–7. IEEE, 2004.

[24] L. Kurth. Xen Wiki - StubDom page. https://wiki.xenproject.org/wiki/StubDom, accessed 2016-11-30.

[25] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009.

[26] J. Levon and P. Elie. Oprofile: A system profiler for linux, 2004.

[27] B. Liu. xen: free_domheap_pages: delay page scrub to idle loop. Xen development mailing list, https://lists.xenproject.org/archives/html/xen-devel/2014-05/msg02436.html, accessed 2016-11-30.

[28] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, et al. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, 2015.

[29] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430. IEEE, 2012.

[30] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *International Conference on Cloud Computing*, pages 115–131. Springer, 2009.

[31] S. Popuri. A tour of the mini-os kernel. https://www.cs.uic.edu/~spopuri/minios.html, accessed 2016-11-30.

[32] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library os from the top down. In *ACM SIGPLAN Notices*, volume 46, pages 291–304. ACM, 2011.

[33] I. Pratt, D. Magenheimer, H. Blanchard, J. Xenidis, J. Nakajima, and A. Liguori. The ongoing evolution of xen. In *Proceedings of the Ottawa Linux Symposium*, 2006.

[34] C. Reiss, J. Wilkes, and J. 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 2016-11/30.

[35] M. Russinovich. Inside windows azure: the cloud operating system. https://channel9.msdn.com/events/Build/BUILD2011/SAC-853T, accessed 2016-11-30, 2011.

[36] K. Rzeszutek Wilk. Xen profiling: Oprofile and perf. https://wiki.xenproject.org/wiki/Xen_Profiling:_oprofile_and_perf, accessed 2016-11-30.

[37] The Netflix Tech Blog. Auto scaling in the amazon cloud. http://techblog.netflix.com/2012/01/autoscaling-in-amazon-cloud.html, accessed 2016-11-30.

[38] V. M. Weaver. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, page 80, 2013.

[39] J. Weinman. Time is Money: The Value of "on-demand". http://www.joeweinman.com/Resources/Joe_Weinman_Time_Is_Money.pdf, accessed 2016-11-30, 2011.

[40] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

[41] J. Wilkes. More Google cluster data. Google research blog, Nov. 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html, accessed 2016-11-30.

[42] S. Yi, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 236–243. IEEE, 2010.

[43] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr. Fast restore of checkpointed memory using working set estimation. In *ACM SIGPLAN Notices*, volume 46, pages 87–98. ACM, 2011.

[44] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite. Optimizing vm checkpointing for restore performance in vmware esxi. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 1–12, 2013.

[45] L. Zhang, J. Litton, F. Cangialosi, T. Benson, D. Levin, and A. Mislove. Picocenter: Supporting long-lived, mostly-idle applications in cloud environments. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 37. ACM, 2016.

[46] J. Zhu, Z. Jiang, and Z. Xiao. Twinkle: A fast resource provisioning mechanism for internet services. In *Proceedings of IEEE INFOCOM*, pages 802–810. IEEE, 2011.