Exploring VM Introspection: Techniques and Trade-offs

Sahil Suneja
University of Toronto
sahil@cs.toronto.edu

Canturk Isci
IBM T.J. Watson Research
canturk@us.ibm.com

Eyal de Lara
University of Toronto
delara@cs.toronto.edu

Vasanth Bala
IBM T.J. Watson Research
vbala@us.ibm.com

Abstract

While there are a variety of existing virtual machine introspection (VMI) techniques, their latency, overhead, complexity and consistency trade-offs are not clear. In this work, we address this gap by first organizing the various existing VMI techniques into a taxonomy based upon their operational principles, so that they can be put into context. Next we perform a thorough exploration of their trade-offs both qualitatively and quantitatively. We present a comprehensive set of observations and best practices for efficient, accurate and consistent VMI operation based on our experiences with these techniques. Our results show the stunning range of variations in performance, complexity and overhead with different VMI techniques. We further present a deep dive on VMI consistency aspects to understand the sources of inconsistency in observed VM state and show that, contrary to common expectation, pause-and-introspect based VMI techniques achieve very little to improve consistency despite their substantial performance impact.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Design studies; D.2.8 [*Metrics*]: Performance measures; D.4.2 [*Storage Management*]: Virtual memory; D.4.1 [*Process Management*]: Synchronization

Keywords Virtualization; Virtual Machine; VMI; Taxonomy; Consistency

1. Introduction

Virtual machine introspection (VMI) [28] has been used to support a wide range of use cases including: digital forensics [13, 14, 24, 31, 63]; touchless systems monitoring-tracking resource usage, ensuring system health and policy compliance [32, 67]; kernel integrity and security monitoring-intrusion detection, anti-malware, firewall and virus scanning [9, 23, 26–28, 34, 36, 38, 54, 64]; cloud management and infrastructure operations such as VM sizing and migration, memory checkpointing and deduplication, device utilization monitoring, cloud-wide information flow tracking and policy enforcement, cluster patch management, and VM similarity clustering [3, 8, 10, 15, 33, 58, 76].

There are different ways in which VMI gains visibility into the runtime state of a VM, ranging from exposing a raw byte-level VM memory view and traversing kernel data structures in it [5, 6, 14, 25, 40, 41, 46], to implanting processes or drivers into the guest [16, 29]. A security-specific survey of the VMI design space can be found in [37]. Although several techniques to expose VM state have been developed independently over the years, there is no comprehensive framework that puts all these techniques in context, and compares and contrasts them. Understanding the trade-offs between the competing alternatives is crucial to the design of effective new applications, and would aid potential VMI users in deciding as to which of the myriad techniques to adopt as per their requirements and constraints.

In this paper, we present a thorough exploration of VMI techniques, and introduce a taxonomy for grouping them into different classes based upon four operation principles: (i) whether guest cooperation is required; (ii) whether the technique creates an exact point-in-time replica of the guest's memory; (iii) whether the guest has to be halted; and, (iv) the type of interface provided to access guest state.

We present the results of a qualitative and quantitative evaluation of a wide range of VMI techniques. We compare software-only methods; VMI techniques that rely on hardware memory acquisition (e.g., System Management Mode (SMM) in x86 BIOS [7, 74], DMA [1, 9, 12, 47, 55], system bus snooping [43, 48]) are beyond the scope of this paper. The qualitative evaluation considers techniques available in

VEE '15, March 14–15, 2015, Istanbul, Turkey..

http://dx.doi.org/10.1145/2731186.2731196

VMware, Xen and KVM. The quantitative evaluation is restricted to a single hypervisor to minimize environmental variability. We use KVM as it has the highest coverage of VMI techniques and gives us more control, but its results can be extrapolated to similar VMware and Xen techniques that fall in the same taxonomy classes. We also present a detailed exploration of the memory consistency aspects of VMI techniques. We show the actual reasons behind potential inconsistency in the observed VM state, and the inability of pause-and-introspect based VMI techniques to mitigate all forms of inconsistency, contrary to common expectation.

Our evaluation reveals that VMI techniques cover a broad spectrum of operating points. Their performance varies widely along several dimensions, such as their speed (0.04Hz to 500Hz), resource consumption on host (34% to 140% for always-on introspection), and overhead on VM's workload (0 to 85%). VMI methods may be available out-of-box on different hypervisors or be enabled by third party libraries or hypervisor modifications, giving the user a choice between easy deployability vs. hypervisor specialization. Furthermore, higher performance may be extracted by modifying the hypervisor or host, yielding a performance vs. host/hypervisor specialization tradeoff. Therefore, application developers have different alternatives to choose from based on their desired levels of latency, frequency, overhead, liveness, consistency, and intrusiveness, constrained by their workloads, use-cases, resource budget and deployability flexibility.

The rest of this paper is organized as follows. Section 2 introduces our VMI taxonomy and groups existing VMI methods based on this taxonomy. Section 3 presents a qualitative evaluation of VMI techniques implemented on VMware, Xen and KVM. Sections 4 presents quantitative evaluation of VMI techniques based on KVM. Section 5 explores VMI state consistency. Section 6 present a summary of our key observations, best operational practices and our experiences with all the explored techniques. Finally, Section 7 offers our conclusions.

2. VMI Taxonomy

We characterize VMI techniques based on four orthogonal dimensions: (i) *Guest Cooperation*, whether the technique involves cooperation from code running inside the guest VM; (ii) *Snapshotting*, whether the technique creates an exact point-in-time replica of the guest's memory; (iii) *Guest Liveness*, whether the techniques halts the guest VM; and (iv) *Memory Access Type*, the type of interface provided to access guest state, which can be either via address space remapping, reads on a file descriptor, or through an interface provided by a VM manager or a debugger.

While there can be arbitrary combinations of these dimensions, in practice only a few are employed. Figure 1's taxonomy shows the specific attribute combinations that can categorize the current implementations for accessing in-VM memory state. Some of these methods are hypervisor exposed, while others are either specialized use cases, or leverage low level memory management primitives, or enabled by third party libraries. The proposed taxonomy is general and hypervisor independent. The rest of this section describes the techniques' functionality.

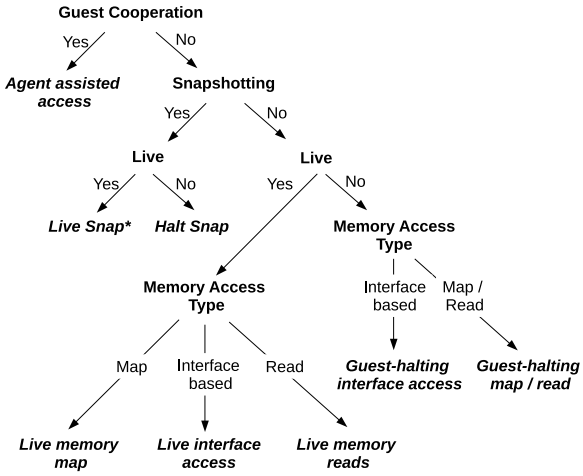


Figure 1: VMI Taxonomy: categorizing current implementations

I. Agent assisted access requiring guest cooperation

These techniques install agents or modules inside the guests to facilitate runtime state extraction from outside.

• **VMWare** `VMSafe()` **[72], XenServer's** `XenGuestAgent` **[17, 18], QEMU's** `qemu-ga` **[57]:** Access VM's memory directly via guest pseudo-devices (`/dev/mem`) or interface with the guest OS via pseudo filesystem (`/proc`) or kernel exported functions. The solutions then communicate either directly through their own custom in-VM agents [2, 22, 69–71], or mediated by the hypervisor [16, 36].

II. Halt Snap

These methods do not require guest cooperation and distinguish themselves for producing a full copy of the guest's memory image, while also pausing the guest to obtain a consistent snapshot.

• **QEMU** `pmemsave`**, Xen** `dump-core`**, Libvirt/Virsh library's** `dump` **and** `save`**, VMWare** `vmss2core`**:** These techniques dump the VM memory to a file. Example usages include Blacksheep [10], Crash [21].

• **QEMU** `migrate` **to file:** Migrates VM to a file instead of a physical host. Essentially similar to memory dumping, but smarter in terms of the content that actually gets written (deduplication, skipping zero pages etc.).

• **LibVMI library's** `shm-snapshot` **[11]**: Creates a VM memory snapshot inside a shared memory virtual filesystem at host. Implemented for both Xen and KVM (QEMU modified). Access to snapshot mediated by LibVMI after internally mapping the memory resident (`/dev/shm/*`) file.

III. Live Snap
These methods obtain a consistent snapshot without pausing the guest.

• HotSnap [20] for QEMU/KVM and similar alternatives for Xen [19, 35, 42, 66, 73] use copy-on-write implementations to create consistent memory snapshots that does not halt the guest. These approaches modify the hypervisor due to lack of default support.

IV. Live Memory Mapping
These methods do not require guest cooperation or a guest memory image capture, and support introspection while the guest continues to run. Methods in this class provide a memory mapped interface to access the guest state.

• **Xen** `xc_map_foreign_range()`**, QEMU Pathogen [60]:** Maps the target guest's memory into the address space of a privileged monitoring or introspection process. Used in libraries such as XenAccess [53] and LibVMI [11], and in cloud monitoring solutions such as IBMon [58] and RTKDSM [32].

• **QEMU large-pages (**`hugetlbfs`**) based and VMWare** `.vmem` **paging file backed VM memory:** Mapping the file that backs a VM's memory into the monitoring or introspection process' address space. Used in OSck [34] for monitoring guest kernel code and data integrity, and in [68] to expose a guest's video buffer as a virtual screen.

• **QEMU and VMWare host physical memory access:** Mapping the machine pages backing the VM's memory, into the introspection process' address space. Leveraging Linux memory primitives for translating the virtual memory region backing the VM's memory inside the QEMU or `vmware-vmx` process (via `/proc/pid/maps` psuedo-file) to their corresponding physical addresses (`/proc/pid/pagemap` file). Not straightforward in Xen, although the administrator domain can access the host physical memory, still need hypervisor cooperation to extract guest backing physical frames list (physical-to-machine (P2M) table).

V. Live Memory Reads
Methods in this class also enable live introspection without perturbing the guest, but access guest state through a file descriptor-based interface.

• **QEMU and VMWare direct VM memory access:** These methods directly read a guest's memory contents from within the container process that runs the VM. This can be achieved in different ways: (i) Injecting a DLL into the `vmware-vmx.exe` container process in VMWare to read its `.vmem` RAM file [45], (ii) Using QEMU's native memory access interface by running the introspection thread inside QEMU itself [8], (iii) Leveraging Linux memory primitives– reading QEMU process' memory pages at the hypervisor (via `/proc/pid/mem` pseudo-file) indexed appropriately by the virtual address space backing the VM memory (`/proc/pid/maps`) [67].

• **LibVMI memory transfer channel:** Requesting guest memory contents over a unix socket based communication channel created in a modified QEMU container process, served by QEMU's native guest memory access interface.

VI. Guest-Halting Memory Map and Reads
These methods achieve coherent/consistent access to the guest memory by halting the guest while introspection takes place (pause-and-introspect), but do not create a separate memory image and access guest memory contents directly. While all the live memory map and read methods can be included in this category by also additionally pausing the guest, we only select one direct read method, employed in literature, as a representative—**QEMU semilive direct access**, that encompasses the guest memory reads (QEMU `/proc/pid/mem`) between `ptrace()`- attach/detach calls. Used in NFM [67] for cloud monitoring under strict consistency constraints.

VII. Live Interface Access
Methods in this class also enable live introspection without perturbing the guest, but access guest state over an interface provided by a third party program.

• **KVM with QEMU monitor's xp [75]**: uses the hypervisor management interface to extract raw bytes at specified (pseudo) physical addresses.

VIII. Guest-Halting Interface Access
Methods in this class halt the guest and access guest state over an interface provided by a third party program.

• **Xen's** `gdbsx`**, VMWare's** `debugStub`**, QEMU's** `gdbserver` GDB stub for the VM: attach a debugger to the guest VM and access guess state over the debugger's interface. This method is used in IVP [62] for verifying system integrity. LibVMI when used without its QEMU patch defaults to using this technique to access guest state. We use the libVMI GDB-access version in our evaluation.

3. Qualitative Comparison

The various VMI techniques described in the previous section follow different operation principles and correspondingly exhibit different properties. Table 1 compares them in terms of the following properties:

- **Guest Liveness:** A live memory acquisition and subsequent VM state extraction is defined in terms of whether or not the target VM continues to make progress normally without any interruption.
- **Memory view consistency:** refers to coherency between the runtime state exposed by the method and the guest's actual state (Section 5).
- **Speed:** How quickly can guest state be extracted with a particular method?
- **Resource consumption on host:** How heavy is a particular approach in terms of the CPU resources consumed by

	Live	View consistency	Speed	Resource cost	VM perf impact	Host and Hypervisor Compatibility		
						Xen	KVM/QEMU	VMWare
Guest cooperation / agent assisted access	✓	✓ (not /dev/mem)	Medium	Medium	Low	VM /dev/mem support or module installation	VM /dev/mem support or module installation	Default; special drivers/tools in VM
Halt Snap		✓	Low	High	High	Default; - In-mem snap via library	Default; - In-mem snap via library + hypervisor modifications	Default
Live Snap	✓	✓	Medium	Low	Low	Hypervisor modifications	Hypervisor modifications	
Live Memory Mapping	✓		Very High	Very Low	Very Low	Default	Hypervisor modifications; - Default file backed mapping with special VM flags, large pages host reservation; - /dev/mem support for host phys mem access	via library; - Default file backed mapping; - /dev/mem support host phys mem access
Live Memory Reads	✓		High	Low	Very Low		Compatible (via /proc); - Mem transfer channel via library + hypervisor mod.	via library
Guest-Halting Memory Map and Reads		✓	Medium	Low	Medium	Compatible (+ guest pause)	Compatible (+ guest pause)	Compatible (+ guest pause)
Live Interface Access	✓		Very Low	Very High	Low		Default (via management interface)	
Guest-Halting Interface Access		✓	Very Low	Very High	Low	Default	Default + special VM initialization flags	Default + special VM config options

Table 1: Qualitative comparison of VMI techniques- empty cells in compatibility column do not necessarily indicate missing functionality in hypervisor.

it, normalized to monitoring 1 VM at 1Hz (memory and disk cost is negligible for all but snapshotting methods).

- **VM performance impact:** How bad does memory acquisition and state extraction hit the target VM's workload.
- **Compatibility:** How much effort does deploying a particular technique cost in terms of its host and hypervisor compatibility- whether available as stock functionality, or requiring hypervisor modifications, or third party library installation, or host specialization.

Table 1 only contrasts these properties qualitatively, while a detailed quantitative comparison follows in the next section. The compatibility columns in the table, do not indicate whether a functionality is available or missing from a hypervisor, rather whether the functionality has been 'shown' to work by virtue of it been exported as a default feature by the hypervisor or via libraries or hypervisor modifications.

As can be seen, no one technique can satisfy all properties at the same time, leading to different tradeoffs for different use cases. One tradeoff is between the conflicting goals of view consistency and guest liveness for almost all techniques. If the user, however, desires both, then he would either have to let go of guest independence by opting for the guest cooperation methods that run inside the guest OS scope, or choose a hardware assisted out-of-band approach using transactional memory [44]. COW based live snapshotting seems to be a good compromise, providing an almost-live and consistent snapshot.

Another tradeoff is between a VMI techniques' performance and generality in terms of requirements imposed on the host's runtime. For example, the live direct-reads method in KVM is sufficiently fast for practical monitoring applications and works out-of-box, still an order of magnitude higher speed can be achieved with live memory-mapping techniques by either enabling physical memory access on host, or reserving large pages in host memory for file-backed method. Although the latter come with a tradeoff of increasing system vulnerability (`/dev/mem` security concerns) and memory pressure (swapping concerns [51, 65]).

4. Quantitative Comparison

To quantitatively compare VMI techniques, we use a simple generic use case of periodic monitoring. This entails extracting at regular intervals generic runtime system information from the VM's memory: CPU, OS, modules, N/W interfaces, process list, memory usage, open files, open network connections and per-process virtual memory to file mappings. This runtime information is distributed into several in-memory kernel data structures for processes (`task_struct`), memory mapping (`mm_ struct`), open files (`files_struct`), and network information (`net_devices`) among others. These `struct` templates are overlaid over the exposed memory, and then traversed to read the various structure fields holding the relevant information [46], thereby converting the byte-level exposed memory view into structured runtime VM state. This translates to reading around 700KB of volatile runtime state from the VM's memory, spread across nearly 100K read/seek calls.

We compare the different VMI techniques along the following dimensions:

1. Maximum frequency of monitoring
2. Resource usage cost on host
3. Overhead caused to the VM's workload

We run different benchmarks inside the VM to measure monitoring's impact when different resource components are stressed - CPU, disk, memory, network and the entire system as a whole. The different targeted as well as full system benchmarks tested are as follows.

1. x264 CPU Benchmark: Measured is x264 video encoding benchmark's [50] (v1.7.0) frames encoded per second.

2. Bonnie++ Disk Benchmark: Measured is bonnie++'s [61] (v1.96) disk read and write throughputs as it processes 4GB of data sequentially. High performance `virtio` disk driver is loaded in the VM, and disk caching at hypervisor is disabled so that true disk throughputs can be measured, which are verified by running `iostat` and `iotop` on host. Host and VM caches are flushed across each of the 5 bonnie++ runs.

3. STREAM Memory Benchmark: Measured is STREAM benchmark's [39] (v5.10) `a[i] = b[i]` sort of in-memory data copy throughput. We modified the STREAM code to also emit the 'average' sustained throughput across all the STREAM iterations (N=2500), along with the default 'best' throughput. The array size is chosen to be the default 10M elements in accordance with STREAM's guidelines of array size vs. cache memory size on the system. The memory throughputs observed inside the VM are additionally confirmed to be similar to when STREAM is run on the host.

4. Netperf Network Benchmark: Measured is network bandwidth when a netperf [59] server (v2.5.0) runs inside the VM, while another physical machine is used to drive TCP data transfer sessions (N=6). High performance `virtio` network driver is loaded in the VM, and the network throughput recorded by the client is confirmed to be similar to when the netperf server runs on the host machine itself.

5. Full System OLTP Benchmark: Measured is Sysbench OLTP database benchmark's [4] (v0.4.12) throughput (transactions per sec) and response time. The benchmark is configured to fire in 50K database transactions, which includes a mix of read and write queries, on a 1M row InnoDB table. Optimal values are ensured for InnoDB's service thread count, cache size and concurrency handling, with the in-VM performance verified to be similar to on-host.

6. Full System Httperf Benchmark: Measured is the incoming request rate that a webserver VM can service without any connection drops, as well as its average and 95th percentile response latency. A 512MB working set workload is setup in a webserver VM, from which it serves different 2KB random content files to 3 different httperf clients (v0.9.0) running on 3 separate machines. The file size is chosen to be 2KB so that server is not network bound.

Experimental Setup: The host is an 8 core Intel Xeon E5472 @ 3GHz machine, with 16GB memory and Intel Vt-x hardware virtualization support. The software stack includes Linux-3.8 host OS with KVM support, Linux 3.2 guest OS, libvirt 1.0.4, QEMU 1.6.2, libvmi-master commit-b01b349 (for in-memory snapshot).

In all experiments except the memory benchmark, the target VM has 1GB of memory and 1 VCPU. Bigger memory impacts snapshotting techniques linearly, without any noticeable impact on other techniques as they are agnostic to VM size. Also, more VCPUs do not affect VMI performance much, except for generating some extra CPU-specific state in the guest OS that also becomes a candidate for state extraction. We select 1 VPCU so as to minimize any CPU slack which could mask the impact of the VMI techniques on the VM's workload. However, in case of the memory benchmark, a multicore VM was necessary as the memory bandwidth was observed to increase with the number of cores, indicating a CPU bottleneck, with the peak bandwidth being recorded on employing 4 cores (almost twice as much as on a singe core; going beyond 4 had no further improvement).

Discussion: (i) We do not include live snapshotting in our quantitative evaluation because of the unavailability of a standalone implementation (patch or library) for our KVM testbed, while its qualitative performance measures are borrowed from [35]. Live snapshotting is expected to have a much better performance as indicated in Table 1. Quantitatively, while monitoring the target VM, live snapshotting is expected to achieve $\sim$5Hz of monitoring frequency, with about 10% CPU consumption on host, and $<$13% hit on the VM's workload [35].

(ii) Also, we do not explicitly compare guest cooperation methods in the remainder of this section. This is because the default `qemu-ga` guest agent implementation on KVM/QEMU is pretty limited in its functionality. Absence of a dynamic `exec` capability with the agent means the generic monitoring process on host has to read all relevant guest `/proc/*` files to extract logical OS-level state [2], which takes about 300ms per transfer over the agent's serial channel interface. This translates to a maximum monitoring frequency of the order of 0.01Hz with $<$1% CPU consumption on host and guest. However, a better way would be for a custom agent to do the state extraction processing in-band and only transfer the relevant bits over to the host, along the lines of [52]. Emulating this with qemu agent, to extract the 700KB of generic VM runtime state, results in a maximum monitoring frequency of the order of 1Hz with about 50% CPU consumption on host, and a 4.5% hit on the VM workload.

4.1 Maximum Monitoring Frequency

Figure 2 compares the maximum attainable frequency at which an idle VM can be monitored while employing the different VMI techniques. The monitoring frequency is calculated from the average running time for 1000 monitoring iterations. We use an optimized version of LibVMI in this study that skips per iteration initialization/exit cycles. Disabling this would add over 150ms latency per iteration thereby lowering the maximum monitoring frequency, most noticeably of the live memory transfer channel implementation.

Interestingly, when sorting the methods in increasing order of their maximum monitoring frequency, each pair of methods shows similar performance, that jumps almost always by an order of magnitude across the pairs. This is be-

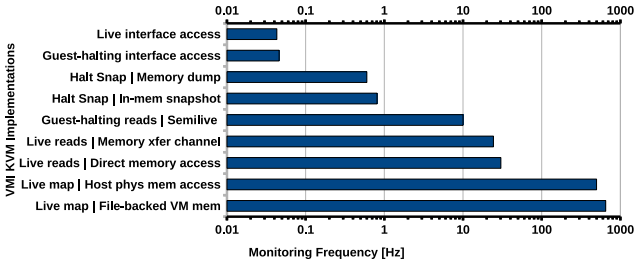


Figure 2: Comparing maximum monitoring frequency across all KVM instances of VMI techniques

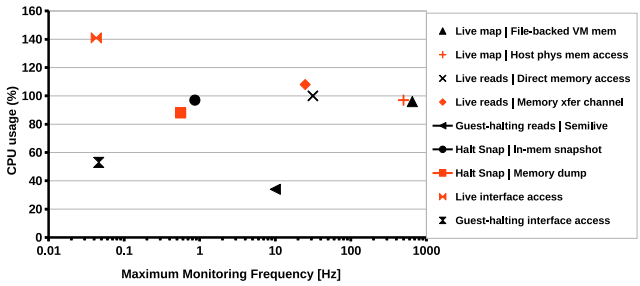


Figure 3: CPU used vs. maximum monitoring frequency

cause the candidates per pair belong to the same taxonomy category, hence they follow similar operation principles, except for interface based methods where the frequency is limited by the interface latency. Amongst the live memory access methods, mapping is much superior to direct reads primarily because of greater system call overheads in the latter (multiple `read()/seek()` vs. single `mmap()`). The next best is guest-halting direct reads that stuns the VM periodically for a few milliseconds, while still being much faster than guest-halting snapshotting methods that halt the VM for a few seconds. Finally, the methods interfacing with the management layer and GDB are the slowest because of yet another layer of indirection.

The maximum monitoring frequencies can vary with the workload inside the VM. Depending upon how active the VM is, it would change the amount of runtime state that exists in the VM, thereby leading to a change in the time required to extract this state. This can easily be observed in the maximum frequencies recorded with httperf in Section 4.4.1 which decrease by 4X due to a proportional increase in runtime state.

4.2 Resource Cost on Host

Monitoring with almost all methods has a negligible space footprint, except for snapshotting techniques that consume space, on disk or memory, equivalent to the VM's size. As for the CPU cost, Figure 3 plots the CPU resource usage on host while an idle VM is monitored at the maximum frequency afforded by each technique. The graph shows the same pairwise grouping of the methods as in the case of their maximum monitoring frequency. The exception here is that the management interface is much heavier than the debugger interface, although both deliver the same monitoring frequency.

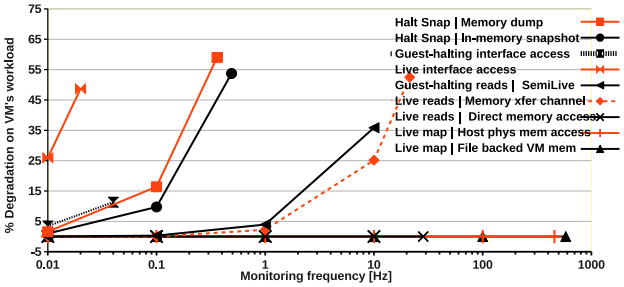
The previous frequency comparison graph showed that the live memory mapping methods were an order of magnitude faster than live direct reads, which were themselves faster than guest-halting reads and snapshotting methods. This graph shows that the **better performance does not come at an added cost** as all of these except for halting-reads consume similar CPU resources. However, with the same CPU consumption, the methods situated more towards the increasing X axis are more efficient in terms of normalized CPU cost per Hz. Hence, amongst the live methods having the same CPU consumption, the higher efficiency of guest memory mapping can be easily observed. Also, the lower CPU usage for the halting-reads method is misleading as the graph does not plot the impact on the VM with each technique. So even though the former can hit the 10Hz frequency while consuming <40% CPU as compared to live reads that consume 100% CPU for about 30Hz, yet it is costlier because it stuns the VM periodically thereby disturbing its workload heavily. The next section quantifies this impact.

4.3 Impact on VM's Performance

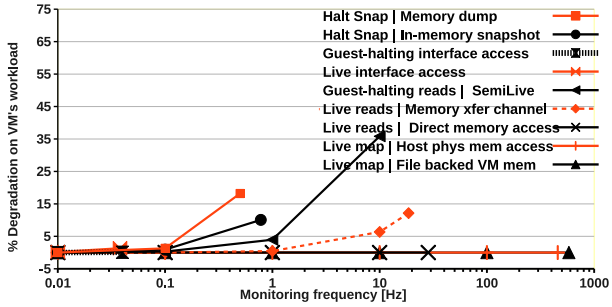
We run targeted workloads inside the VM stressing different resource components, and measure the percentage overhead on their corresponding performance metrics. VM impact is measured for the lowest monitoring frequency of 0.01 Hz, increasing in orders of 10 up to 10Hz or the maximum attainable frequency for each VMI technique. Each benchmark is run enough number of times to ensure sufficient monitoring iterations are performed for each method at each frequency. The graphs only plot the mean values while the error bars are omitted for readability (the experimental variation was within 5% of the means). We use the guest reported benchmark performance metrics, after having ensured that the guest timing matches with the host timings throughout the benchmarks' progress.

In the experiments, our VMI based monitoring application runs on a separate host core, while we experiment with two different configurations mapping the VM's VCPU to host's PCPU[1]. In the first 1VPCU-1PCPU configuration, we pin to a single core on host the QEMU process that runs the main VM thread and all other helper threads that get spawned to serve the monitoring process' memory access requests. In the second 1VCPU-2PCPU configuration, we taskset the QEMU process to two cores on host, the VM still having only one virtual core to itself . We do this to visualize the kind of overheads that would be seen if each technique was given unbounded CPU resources (a single extra core suffices, going beyond this has no additional effect).

[1] The hardware architecture influences the introspection application's as well as the VM's VPCPU-PCPU core mapping. The chosen configuration ensures the least impact on VM due to introspection.



(a) 1VCPU- 1PCPU



(b) 1VCPU - 2PCPU

Figure 4: Comparing % degradation on x264 benchmark's frames-encoded/s as a function of monitoring frequency.

4.3.1 CPU Benchmark

Figure 4(a) plots the percentage degradation on x264's [50] frames encoded per second as a function of monitoring frequency for each technique. The rightmost points for each curve show the overheads for the maximum attainable monitoring frequency for each method. Each data point is obtained by averaging 10 x264 runs.

As can be seen, there is minimal overhead on x264's framerate with the live methods (except the libVMI memory transfer channel implementation), while for the rest, the overhead decreases with decreasing monitoring frequency. Biggest hit is observed for methods that quiesce the VM, as expected.

Figure 4(b) compares x264's performance degradation when each technique is given unbounded CPU resources in the 1VCPU - 2PCPU taskset configuration. As a result, the VM overhead is greatly reduced for methods that spawn QEMU threads to extract VM state, as the main QEMU thread servicing the VM now no longer has to contend for CPU with the other helper threads that get spawned to serve the monitoring process' memory access requests. The halting-read method, which wasn't using a full CPU to begin with, has no use for the extra CPU resources and thus the VM overhead remains the same owing to the periodic VM stuns.

This is the only case where we compare the performance of all candidate techniques. Our main focus is actually on how the categories themselves compare in terms of performance degradation of the target VM's workload. Hereafter, we only present results for one representative method from each category- namely memory dumps (guest-halting snapshotting), management interface (interface access), semilive direct access (halting-reads), QEMU direct memory access (live memory reads), and file-backed VM memory (live memory map). Although not explicitly shown, the omitted methods follow performance trends similar to their sibling candidates from the same taxonomy category. The interface access methods are also observed to exhibit similar performance.

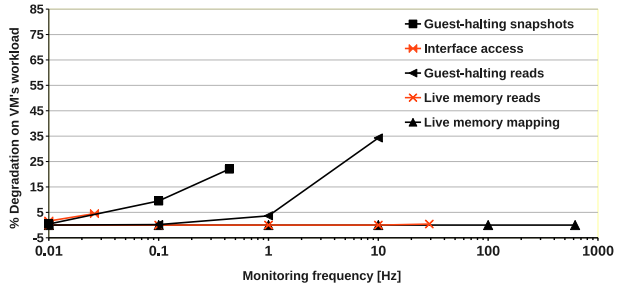
4.3.2 Memory, Disk and Network Benchmarks

Figure 5 plots the impact on the VM's memory, disk and network throughputs, owing to VMI based monitoring. Impact is mostly observed for only the methods that quiesce the VM, and does not improve markedly when extra CPU resources (1VPCU-2PCPU mapping) are provided to the techniques. This is because the CPU is not the bottleneck here, with the workloads either being limited by the memory subsystem, or bounded by network or disk IO.

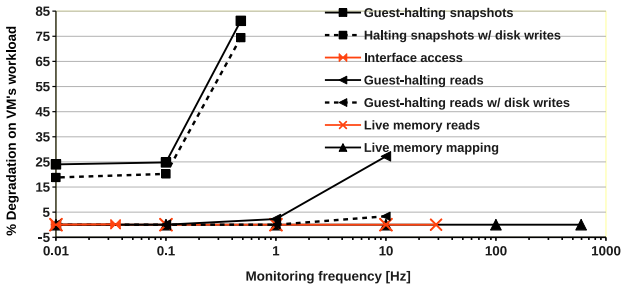
The degradation on STREAM [39] benchmark's default 'best' (of all iterations) memory throughput was negligible even while monitoring with methods that quiesce the VM. However, the techniques' true impact can be seen in Figure 5(a) that compares the percentage degradation on STREAM's 'average' (across all iterations) memory throughput. In other words, the impact is only observed on the sustained bandwidth and not the instantaneous throughput.

For the impact on bonnie++ [61] disk throughputs, separate curves for disk writes are only shown for VM quiescing methods (Figure 5(b)), the rest being identical to those of reads, with the main noticeable difference being the minimal impact seen on the write throughput even with the halting-reads method. This can be attributed to the fact that the VM's CPU is not being utilized at its full capacity and spends a lot of time waiting for the disk to serve the write requests made from bonnie++. Hence, minor VM stunning doesn't hurt the benchmark so bad, as the work gets delegated to the disk. This, along with the writeback caching in the kernel, also means that the worst-case per-block write latency (not shown in the graphs) does not see a big hit even for methods that quiesce the VM, while their worst-case read latency is an order of magnitude higher.

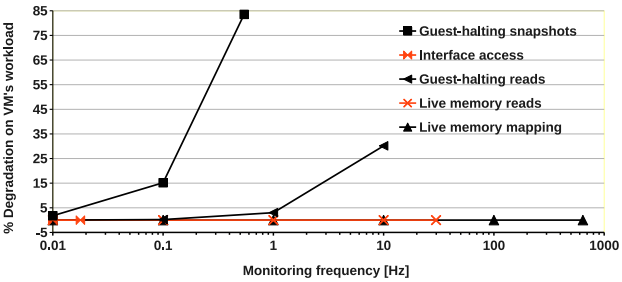
Another interesting observation is the markedly high impact on the disk throughputs with memory dumping, as compared to the CPU intensive benchmark, which moreover shows no improvement even when the monitoring frequency is reduced from 0.1Hz to 0.01Hz. Netperf's [59] network bandwidth also sees a similar hit with guest-halting snapshotting (Figure 5(c)), with its impact curves being very similar to those of the disk (read) throughputs. The difference in this case is that that the overhead curve does not plateau out and eventually subsides to minimal impact at 0.01Hz. As demonstrated later in Section 4.4.1, these high overheads can



(a) Impact on STREAM benchmark's memory copy throughput



(b) Impact on bonnie++'s disk throughputs. Differing behaviour on writes shown separately.



(c) Impact netperf's network transfer bandwidth

Figure 5: Comparing % degradation on memory, disk and network throughput as a function of monitoring frequency

attributed to the backlog of pending IO requests that dumping (and hence VM quiescing) creates in the network and disk IO queues.

4.4 Real Workload Results

After characterizing VMI based monitoring's impact on individual VM resources, we use two full system benchmarks to understand the impact on real world deployments-database and webserver. We omit the OLTP database benchmark [4] for brevity, as the graphs for impact on its transaction throughput and response times are pretty much identical to the disk read benchmark, being attributed to the backlogging in the database transaction queues. Instead we dig deep into the httperf benchmark to inspect these queue perturbations.

4.4.1 Full System Httperf Benchmark

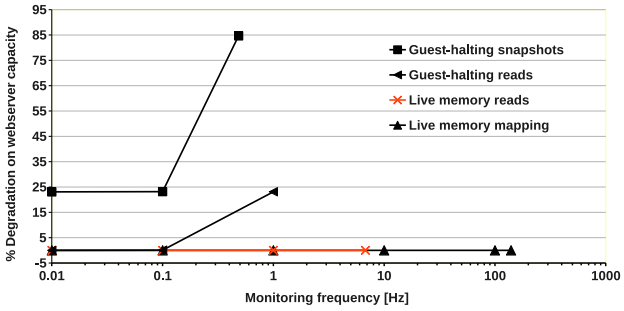
Figure 6(a) plots the impact on the webserver VM's sustainable request rate as compared to the base case without any monitoring, for the different VMI techniques under different monitoring frequencies. Each data point in the graph is obtained by averaging 3 httperf [49] runs, each run lasting for 320s.

Amongst all the benchmarks, httperf is hit the worst by methods that quiesce the VM, even at low monitoring frequency, with the halting-reads method recording ∼25% impact even at 1Hz. With memory dumping, like in case of the disk and OLTP benchmarks, the impact on the sustainable request rate is not lowered even with extra CPU resources afforded to the QEMU process, as well as when the monitoring frequency is reduced from 0.1Hz to 0.01Hz. We explain this with an experiment later in this Section.

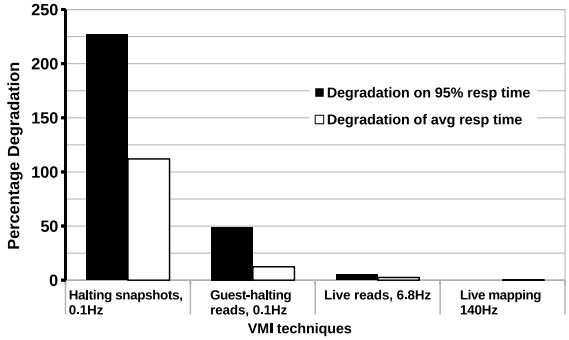
Also note the much lower maximum monitoring frequencies recorded for the different techniques as they monitor the httperf workload. The longer monitoring cycles are because the amount of state extracted is far more than other benchmarks (∼4X), owing to several `apache` processes running inside the VM. This also prevents the interface based approaches from operating even at 0.01Hz, while the halting-reads method is unable to operate at its usual 10Hz (iteration runtime ∼150ms).

The sustainable request rate is only one half of the story. Figure 6(b) also plots the impact on the webserver VM's average and 95th percentile response latencies. Shown are overheads for the practical monitoring frequencies of 0.1Hz for techniques that quiesce the VM, and for maximum attainable monitoring frequencies for the other live methods. As can be seen, even if a user was willing to operate the webserver at 75% of its peak capacity, while snapshotting once every 10s for view consistent introspection, they should be aware of the fact that the response times would shoot up 100% on an average, going beyond 200% in the worst case. The particular requests experiencing these massive degradations can be spotted in a server timeline graph, omitted for brevity, where the response times jump quite a bit for about 50s after a single dumping iteration (<2s).

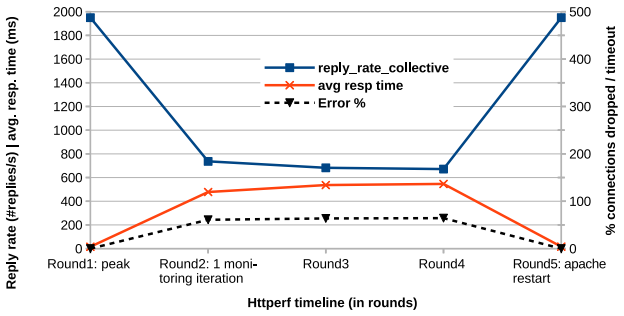
Finally, we investigate why guest-halting snapshotting shows a horizontal impact curve from 0.1Hz to 0.01Hz in Figure 6(a), instead of the impact on server's capacity lowering on snapshotting it 10 times less frequently. As discussed above, when the webserver operates at 75% of its peak capacity (serving 1500 requests/s as opposed to 1950), the jump in response times eventually subsides after a single snapshotting iteration, and no requests are dropped. If the requests arrive at any rate greater than this, it is observed that a single <2s dumping cycle degrades the server capacity to ∼700 serviced requests/s, with several connection drops, and the server doesn't recover even after 15 minutes. Figure 6(c) visualizes this observation for 5 httperf rounds of 320s each, plotting the (i) server capacity (`reply_rate`) (ii) avg. response time per request, and (iii) percentage connections dropped (`Error %`). In the end, the server has to be 'refreshed' with an `apache` process restart to clear up



(a) Impact on httperf's sustainable request rate



(b) Impact on httperf's response times



(c) httperf 1950 req/s + 1 round of memory dumping

Figure 6: Comparing % degradation on httperf's metrics as a function of monitoring frequency

all the wait queues to bring it back up to its base capacity. Hence, because the server is operating at its peak capacity in this case, the wait queues are operating at a delicate balance with the incoming request rate. Any perturbation or further queuing introduced by a single VM quiescing cycle destroys this balance, thereby creating a backlog of pending requests which the webserver never seems to recover from. The behaviour is same for any incoming rate >1500 requests/s. And this is why even for 0.01Hz monitoring frequency, the server can only handle 1500 requests per sec at best. Note that the measurements are made from the clients' side in httperf, so the requests from a new round also get queued up behind the pending requests from an earlier round. Hence, from the clients' perspective, a possible eventual server capacity recovery is not observed without a complete server queue flush.

5. Consistency of VM State

A key concern with VMI techniques is the consistency of the observed VM state. Particularly, introspecting a live system while its state is changing may lead to inconsistencies[2] in the observed data structures. An inconsistency during introspection may cause the monitoring process to fail, trying to access and interpret non-existent or malformed data. A common approach to mitigate inconsistencies is to pause/quiesce[3] the systems during introspection (halting-reads method). This is considered a safe approach as the system does not alter its state while the data structures are interpreted [11, 28, 30, 44, 56]. Therefore it is commonly employed for "safe" introspection despite its high overheads as we had shown in the prior sections. In this section we present a deeper exploration of what these inconsistencies are, their likelihood, and when pause-and-introspect (PAI) solutions help. Our investigation leads to some interesting key observations. First, we show that there are multiple forms of inconsistencies, both in intrinsic VM state and extrinsic due to live introspection. Second, contrary to common expectation, PAI does not mitigate all forms of inconsistency.

5.1 Inconsistency Types

We capture inconsistencies by recording the `read()` or `seek()` failures in the introspection process, while the VM being monitored runs workloads (Section 5.2) that continuously alter system state. Each of these failures denote an access to a malformed or non-existent data structure. Furthermore, by tracing back the root of these failures, we were also able to categorize every inconsistency observed as follows. We further verified the exact causes of each inconsistency occurrence by running Crash [21] on a captured memory snapshot of the paused VM under inconsistency.

I. Intrinsic Inconsistencies

This category of inconsistencies occur due to different but related OS data structures being at inconsistent states themselves—for a short period—in the OS, and not because of live introspection. These inconsistencies still persist even if PAI techniques are employed instead of live introspection. We subcategorize these into the following types:

I.A **Zombie Tasks:** For tasks marked as dead but not yet reaped by the parent, only certain basic `task_struct` fields are readable. Others such as memory mapping information, open files and network connections lead to inconsistency errors when accessed.

I.B **Dying Tasks:** For tasks that are in the process of dying but not dead yet (marked "exiting" in their `task_struct`), their memory state might be reclaimed by the OS. Therefore, although their state seems still available, accessing

[2] While the OS itself is not inconsistent, the observed inconsistencies arise because of a missing OS-context within VMI scope.

[3] We use pause/quiesce/halt to refer to the same guest state; not to be confused with the possibly different interpretations from the point of view of the OS.

these can lead to NULL or incorrect values being read by the monitoring process.

I.C **As-good-as-dead tasks:** We still optimistically go ahead and extract state for tasks of the previous type - tagged as exiting. We skip them only in cases where the memory info data structure `mm_struct` is already NULL which means not only are this task's memory mappings unavailable, but any attempt to extract its open files / network connections list is also highly likely to fail.

I.D **Fresh tasks:** For newly-created processes, all of their data structures do not instantaneously get initialized. Therefore, accessing the fields of a fresh process may lead to transient `read()` errors, where addresses read may be NULL or pointing to incorrect locations.

II. Extrinsic Inconsistencies

This second category of inconsistencies occur during live introspection and only these can be mitigated by PAI techniques. The reason for these inconsistencies is VM state changing during introspection. We subcategorize these into the following types:

II.A **Task Dies During Monitoring:** For tasks that die while their data structures were being interpreted, data fields and addresses read after the task state is recycled lead to `read()/seek()` errors.

II.B **Attributes Change During Monitoring:** In this case, while the tasks themselves keep alive, their attributes that point to other data structures might change, such as open files, sockets or network connections. In this case accessing these data structures based on stale/invalid memory references leads to inconsistency errors.

5.2 Quantitative Evaluation

We first create a benchmark, *cork*, that rapidly changes system state by forking and destroying processes at various rates, and use it with a process creation rate of 10Hz and a process lifetime of 1s. We quantify the occurrence probabilities of inconsistencies with two workloads: (i) our simple *cork* benchmark, which stresses the process create/delete dimension; and (ii) a webserver at its peak capacity serving incoming HTTP requests from three separate *httperf* clients for 2^{18} different 2KB files, which stresses both the process and file/socket open/close dimensions.

Figure 7 shows the observed probabilities for all the different inconsistency types for both benchmarks. These probabilities are computed from 3 separate runs, each of which repeat 10,000 introspection iterations (based on the live direct memory read approach) while the benchmarks are executed. The observed results are independent of the introspection frequency. As the figure shows, most inconsistencies are rather rare events (except for one corner case with *httperf*), and the majority of those observed fall into category I. While not shown here, when we perform the same experiments **with the halting-reads approach, all dynamic state extrinsic inconsistencies of Category II disappear, while Category I results remain similar**.

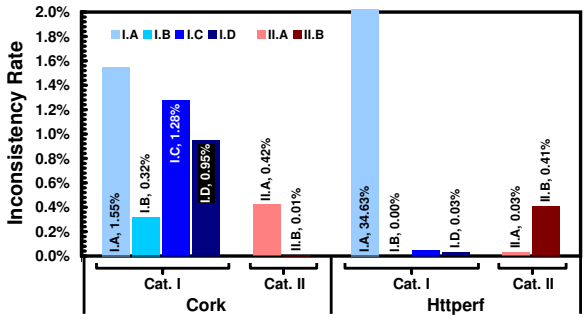


Figure 7: Observed inconsistency probabilities for all categories.

The quantitative evaluation shows some interesting trends. First, we see that Category II events are rather rare (less than 1%) even for these worst-case benchmarks. Therefore, for most cases PAI techniques produce limited return on investment for consistency. If strong consistency is what is desired regardless of cost, then PAI approaches do eliminate these dynamic state inconsistencies. The cost of this can be up to 35% with a guest-halting direct-reads approach for 10Hz monitoring, and 4% for 1Hz monitoring, in terms of degradation on VM's workload. *Cork* records more type $II.A$ inconsistencies, whereas the webserver workload exhibits more of type $II.B$. This is because of the continuous closing and opening of files and sockets, while serving requests in the webserver case. Both of these, however, occur infrequently—in only 0.4% of the iterations. *Cork* also exhibits type $I.C$ and $I.D$ inconsistencies for freshly created and removed tasks, as the OS context itself becomes temporarily inconsistent while updating task structures. One unexpected outcome of this evaluation is the very high rate of type $I.A$ inconsistencies with the webserver, which also has a significant occurrence in *cork*. The amount of time state is kept for zombie tasks varies by both system configuration and load, and can lead to substantial VMI errors (type $I.A$ inconsistency) as seen with the webserver. Zombies are alive until the parent process reads the child's exit status. If the parent process dies before doing so, then the system's `init` process periodically reaps the zombies. Under high loads, the webserver forks several `apache` worker threads and it takes a while before reaping them, thereby leading to their zombie state existence for longer durations.

6. Observations and Recommendations

In this section, we summarize our observations and present our suggestions to VMI users in selecting the technique best suited to their requirements and constraints.

• **Broad Spectrum of Choices:** There are several available VMI alternatives operating on different principles ranging from dumping to memory-mapping. Their performance varies widely along several dimensions such as their speed, resource consumption, overhead on VM's workload, view

consistency, and more. These methods may be available out-of-box on different hypervisors or be enabled by third party libraries or hypervisor modifications, giving the user a choice between easy deployability vs. hypervisor specialization.

• **Guest Cooperation vs. Out-of-band:** If the user has sufficient resources allocated to his VMs, and installing in-VM components is acceptable, then guest-cooperation is a great way of bridging VMI's semantic gap. If this isn't acceptable, or security and inaccuracy of in-VM entities is an additional concern, then, the out-VM methods are a good alternative. The latter also helps against vendor lock-in, if the user prefers uninterrupted functionality with VM mobility across hypervisors without specializing his VMs for a particular hypervisor.

• **VMI use-case:** Some techniques are more suitable in certain scenarios. For example, high speed live methods are best for high frequency realtime monitoring such as process level resource monitoring, continuous validation, best effort security monitoring. On the other hand, snapshotting techniques are useful when all that is needed is an (infrequent) point in time snapshot, as in digital forensics investigation. For infrequent peeking into guest memory, a simple management interface access would suffice, while for guest debugging or crash troubleshooting, the guest-halting GDB-access interface would be the most suitable to freeze and inspect the guest in its inconsistent state without any regards to performance or overhead. Where strict view consistency is desired within acceptable overhead, guest-halting memory mapping/reads would work well such as for low frequency security scanning and compliance audits. Low frequency monitoring offers a lot more flexibility in terms of the choice of technique, except if the workloads are bound by specific resources as discussed next.

• **VM Workload:** Along with the intended VMI use-case, the target VM's workload can also influence the choice of introspection technique. If the user's workload is not bound by a particular VM resource, then there is more flexibility in selecting the introspection technique as well as its speed (frequency), even the ones that quiesce the VM. Even if it is CPU-intensive or memory bound, it can still tolerate guest-halting snapshotting better than if it were IO bound (disk / network / transactions), because the latter would be more sensitive to perturbation of the service queues, in which case snapshotting can be heavy even at very low monitoring frequencies. On the other hand, IO bound workloads can tolerate the lighter stuns of the guest-halting direct-reads method better than CPU intensive workloads, because the work gets handed off to other components while the CPU halts temporarily. But the halting-reads method's execution length, and hence the VM stun duration, depends on the amount of state to be extracted. So it might not be a good fit on an active VM with rapidly changing state (see rapidly spawning apache processes in httperf evaluation in Section 4.4.1), or an application that accesses large memory such as virusscan.

• **Host/Hypervisor Specialization:** Different hypervisors support different techniques out-of-box, some faster than others (comparison across techniques, not across hypervisors). If the user has freedom of choice over hypervisor selection, e.g. if they are not vendor locked to a particular provider or constrained by enterprise policies, then they may choose the one offering the best technique- fastest or cheapest (resource consumption). Otherwise, if the hypervisor selection is fixed, but the user still has control over the host resources or is willing to modify the hypervisor or install third party libraries, they can further optimize the available option to extract the best performance. For example, a 'direct memory access' method in KVM is sufficiently fast for practical monitoring applications and works out-of-box, still an order of magnitude higher speed can be achieved by either modifying QEMU, or enabling physical memory access on host, or reserving large pages in host memory for file-backed method. Although the latter come with a tradeoff of increasing system vulnerability and memory pressure. This work also shows that libraries or hypervisor modification may not be needed to extract high performance, as depicted by the QEMU direct access live method (enabled by leveraging Linux memory primitives) being more efficient than the LibVMI library's live transfer channel implementation, while the latter also requiring QEMU modifications.

• **Mapping over direct reads:** Amongst the various methods compared in this study, the live methods are the best performing across several dimensions. Amongst these, guest memory mapping is much superior to direct memory reads (e.g. speed order of 100Hz vs 10Hz), primarily because of greater system call overheads in the latter (multiple read()/seek() vs. single mmap()). However, the previous observation's speed vs. hypervisor specialization tradeoff holds true here as well, atleast for KVM.

• **Guest-halting map/reads over snapshotting:** For strict view-consistent monitoring and other VM-snapshot based use-cases, it is better to use halting-reads than halting-snapshot based approaches, because although both techniques quiesce the target VM, the impact on the VM's workload is generally much lower with the former technique, and especially bearable for low monitoring frequency. Also, as shown in experiments, guest-halting snapshotting methods create backlogs in work queues thereby heavily impacting performance. Live snapshotting, on the other hand, is a much better alternative as indicated in Section 3's qualitative analysis and towards the end of Section 4 (as *Discussion*).

• **Consistency vs. Liveness, Realtimeness, and VM performance:** For almost all techniques, view consistency and guest liveness are conflicting goals. If the user, however, desires both, then they would either have to let go of guest independence by opting for the guest cooperation methods that run inside the guest OS scope, or choose a hardware assisted

out-of-band approach using transactional memory [44]. One compromise option is COW snapshotting that provides an almost-live and consistent snapshot.

For the common non-live pause-and-introspect (PAI) based techniques (halting-reads), its maximum monitoring frequency can never equal live's because that would mean the VM is paused all the time and is thus making no meaningful progress. Thus, for PAI techniques, there exists a consistency vs. realtimeness tradeoff in addition to the consistency vs. VM performance tradeoff, the latter evident with high VM overheads with halting-reads.

Consistency Fallacy: Furthermore, as our experiments indicate, PAI techniques, employed for "safe" introspection despite their high VM performance impact, do not mitigate all forms of inconsistency, which are very rare to begin with. There is thus a need to synchronize with the guest OS to determine guest states safe for introspection.

• **Monitoring Overhead vs. Resource Usage:** In the KVM/QEMU implementations of guest-halting snapshotting and interfaced based memory access methods, there exists a tradeoff between the resources available for monitoring versus the impact on the VM being monitored, except for when the target VM has CPU slack. This tradeoff does not hold true for the Live memory map/reads which already have negligible VM overhead in the base case, as well as the halting-reads method that doesn't consume a full CPU to begin with, while the overhead stems from periodic VM stuns.

• **Scalability of approaches:** If the user targets several VMs to be monitored at once, another important metric to consider is scalability. Although an explicit comparison is omitted for brevity, it is relatively straightforward to correlate a technique's maximum frequency with CPU usage, and observe that the live memory map/read techniques all consuming more or less a single CPU core on host would monitor the maximum number of VMs at 1 Hz (ranging between 30 to 500 VMs per dedicated monitoring core).

7. Conclusion

We presented a comparative evaluation of exisiting VMI techniques to aid VMI users in selecting the approach best suited to their requirements and constraints. We organized existing VMI techniques into a taxonomy based upon their operational principles. Our quantitative and qualitative evaluation reveals that VMI techniques cover a broad spectrum of operating points. We show that there is substantial difference in their operating frequencies, resource consumption on host, and overheads on target systems. These methods may be available out-of-box on different hypervisors or can be enabled by third party libraries or hypervisor modifications, giving the user a choice between easy deployability vs. hypervisor specialization. We also demonstrate the various forms of intrinsic and extrinsic inconsistency in the observed VM state, and show that pause-and-introspect based techniques have marginal benefits for consistency, despite their prohibitive overheads. Therefore application developers have different alternatives to choose from based on their desired levels of latency, frequency, overhead, consistency, intrusiveness, generality and practical deployability. We hope that our observations can benefit the community in understanding the trade-offs of different techniques, and for making further strides leveraging VMI for their applications.

Acknowledgments

We would like to thank our anonymous reviewers and our shepherd Kenichi Kourai for their helpful suggestions on improving this paper. We also thank Hao Chen for his insight during the initial phase of this work. This work is supported by an IBM Open Collaboration Research award.

References

[1] Adam Boileau. Hit by a Bus: Physical Access Attacks with Firewire. *RuxCon* 2006. `www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf`.

[2] Adam Litke. Use the Qemu guest agent with MOM. `http://https://aglitke.wordpress.com/2011/08/26/use-the-qemu-guest-agent-with-memory-overcommitment-manager/`.

[3] F. Aderholdt, F. Han, S. L. Scott, and T. Naughton. Efficient checkpointing of virtual machines using virtual machine introspection. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 414–423, May 2014.

[4] Alexey Kopytov. SysBench Manual. `http://sysbench.sourceforge.net/docs/#database_mode`.

[5] Anthony Desnos. Draugr - Live memory forensics on Linux. `http://code.google.com/p/draugr/`.

[6] M. Auty, A. Case, M. Cohen, B. Dolan-Gavitt, M. H. Ligh, J. Levy, and A. Walters. Volatility - An advanced memory forensics framework. `http://code.google.com/p/volatility`.

[7] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[8] M. B. Baig, C. Fitzsimons, S. Balasubramanian, R. Sion, and D. Porter. CloudFlow: Cloud-wide policy enforcement using fast VM introspection. In *IEEE Conference on Cloud Engineering IC2E 2014*, 2014.

[9] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Trans. Dependable Secur. Comput.*, 8(5):670–684, Sept. 2011.

[10] A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM Conference on Com-*

puter and Communications Security, CCS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[11] Bryan Payne. LibVMI Introduction: Vmitools, An introduction to LibVMI. `http://code.google.com/p/vmitools/wiki/LibVMIIntroduction`.

[12] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[13] A. Case, A. Cristina, L. Marziale, G. G. Richard, and V. Roussev. Face: Automated digital evidence discovery and correlation. *Digit. Investig.*, 5:S65–S75, Sept. 2008.

[14] A. Case, L. Marziale, and G. G. RichardIII. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7, Supplement(0):S32 – S40, 2010.

[15] J.-H. Chiang, H.-L. Li, and T.-c. Chiueh. Introspection-based memory de-duplication and migration. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 51–62, New York, NY, USA, 2013. ACM.

[16] T.-c. Chiueh, M. Conover, and B. Montague. Surreptitious deployment and execution of kernel agents in windows guests. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 507–514, Washington, DC, USA, 2012. IEEE Computer Society.

[17] Citrix. Citrix XenServer 6.2.0 Virtual Machine User's Guide. `http://support.citrix.com/servlet/KbServlet/download/34971-102-704221/guest.pdf`.

[18] Citrix Systems Inc. XenServer Windows PV Tools Guest Agent Service. `https://github.com/xenserver/win-xenguestagent`.

[19] P. Colp, C. Matthews, B. Aiello, and A. Warfield. Vm snapshots. In *Xen Summit*, 2009.

[20] L. Cui, B. Li, Y. Zhang, and J. Li. Hotsnap: A hot distributed snapshot system for virtual machine cluster. In *LISA*, 2013.

[21] David Anderson. White Paper: Red Hat Crash Utility. `people.redhat.com/anderson/crash_whitepaper/`.

[22] Dell Quest/VKernel. Foglight for Virtualization. quest.com/foglight-for-virtualization-enterprise-edition/.

[23] B. Dolan-Gavitt, B. Payne, and W. Lee. Leveraging forensic tools for virtual machine introspection. Technical Report GT-CS-11-05, Georgia Institute of Technology, 2011.

[24] J. Dykstra and A. T. Sherman. Acquiring forensic evidence from infrastructure-as-a-service cloud computing: Exploring and evaluating tools, trust, and techniques. *Digital Investigation*, 9:S90–S98, 2012.

[25] Emilien Girault. Volatilitux- Memory forensics framework to help analyzing Linux physical memory dumps. `http://code.google.com/p/volatilitux/`.

[26] Y. Fu and Z. Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *IEEE Security&Privacy'12*.

[27] L. Garber. The challenges of securing the virtualized environment. *Computer*, 45(1):17–20, 2012.

[28] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS*, pages 191–206, 2003.

[29] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 147–156. IEEE, 2011.

[30] B. Hay, M. Bishop, and K. Nance. Live analysis: Progress and challenges. *Security & Privacy, IEEE*, 7(2):30–37, 2009.

[31] B. Hay and K. Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, 2008.

[32] J. Hizver and T.-c. Chiueh. Real-time deep virtual machine introspection and its applications. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 3–14, New York, NY, USA, 2014. ACM.

[33] J. Hizver and T. cker Chiueh. Automated discovery of credit card data flow for pci dss compliance. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 51–58, Oct 2011.

[34] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *ASPLOS*, pages 279–290, 2011.

[35] K.-Y. Hou, M. Uysal, A. Merchant, K. G. Shin, and S. Singhal. Hydravm: Low-cost, transparent high availability for virtual machines. Technical report, HP Laboratories, Tech. Rep, 2011.

[36] A. S. Ibrahim, J. H. Hamlyn-Harris, J. Grundy, and M. Almorsy. CloudSec: A security monitoring appliance for Virtual Machines in IaaS cloud model. In *NSS '11*, pages 113–120.

[37] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion. SoK: Introspections on Trust and the Semantic Gap . In *35th IEEE Symposium on Security and Privacy S&P*, 2014.

[38] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In *CCS '07*, pages 128–138.

[39] John D. McCalpin. Memory Bandwidth: Stream Benchmark. `http://www.cs.virginia.edu/stream/`.

[40] N. L. P. Jr., A. Walters, T. Fraser, and W. A. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197 – 210, 2006.

[41] I. Kollar. Forensic RAM dump image analyser. Master's Thesis, Charles University in Prague, 2010. `hysteria.sk/~niekt0/fmem/doc/foriana.pdf`.

[42] 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 *EuroSys*, 2009.

[43] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Proceedings of the*

22Nd USENIX Conference on Security, SEC'13, pages 511–526, Berkeley, CA, USA, 2013. USENIX Association.

[44] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *HPCA 2014*, 2014.

[45] Marco Batista. VMInjector: DLL Injection tool to unlock guest VMs. `https://github.com/batistam/VMInjector`.

[46] Mariusz Burdach. Digital forensics of the physical memory. 2005. `http://forensic.seccure.net/pdf/mburdach_digital_forensics_of_physical_memory.pdf`.

[47] Maximillian Dornseif. 0wned by an iPod. *PacSec Applied Security Conference* 2004. md.hudora.de/presentations/firewire/PacSec2004.pdf.

[48] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 28–37, New York, NY, USA, 2012. ACM.

[49] D. Mosberger and T. Jin. httperf - a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[50] OpenBenchmarking/Phoronix. x264 Test Profile. `http://openbenchmarking.org/test/pts/x264-1.7.0`.

[51] Oracle's Linux Blog. Performance Issues with Transparent Huge Pages. `https://blogs.oracle.com/linux/entry/performance_issues_with_transparent_huge`.

[52] oVirt. oVirt guest agent. `http://www.ovirt.org/Category:Ovirt_guest_agent`.

[53] B. Payne, M. de Carbone, and W. Lee. Secure and Flexible Monitoring of Virtual Machines. In *Twenty-Third Annual Computer Security Applications Conference*, pages 385 –397, 2007.

[54] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 233–247, 2008.

[55] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[56] J. Pfoh, C. Schneider, and C. Eckert. A formal model for virtual machine introspection. In *Proceedings of the 1st ACM workshop on Virtual machine security*, 2009.

[57] QEMU. Features/QAPI/GuestAgent. `http://wiki.qemu.org/Features/QAPI/GuestAgent`.

[58] A. Ranadive, A. Gavrilovska, and K. Schwan. Ibmon: monitoring vmm-bypass capable infiniband devices using memory introspection. In *HPCVirt*, pages 25–32, 2009.

[59] Rick Jones . Netperf Homepage. `http://www.netperf.org/netperf/`.

[60] A. Roberts, R. McClatchey, S. Liaquat, N. Edwards, and M. Wray. Poster: Introducing pathogen: a real-time virtual-machine introspection framework. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 1429–1432, New York, NY, USA, 2013. ACM.

[61] Russell Coker. Bonnie++. `http://www.coker.com.au/bonnie++/`.

[62] J. Schiffman, H. Vijayakumar, and T. Jaeger. Verifying system integrity by proxy. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 179–200, Berlin, Heidelberg, 2012. Springer-Verlag.

[63] A. Schuster. Searching for processes and threads in microsoft windows memory dumps. *Digit. Investig.*, 3:10–16, Sept. 2006.

[64] A. Srivastava and J. Giffin. Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In *RAID*, pages 39–58, 2008.

[65] Structured Data. Transparent Huge Pages and Hadoop Workloads. `http://structureddata.org/2012/06/18/linux-6-transparent-huge-pages-and-hadoop-workloads/`.

[66] M. H. Sun and D. M. Blough. Fast, lightweight virtual machine checkpointing. Technical report, Georgia Institute of Technology, 2010.

[67] S. Suneja, C. Isci, V. Bala, E. de Lara, and T. Mummert. Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 249–261, New York, NY, USA, 2014. ACM.

[68] Toby Opferman. Sharing Memory with the Virtual Machine. `http://www.drdobbs.com/sharing-memory-with-the-virtual-machine/184402033`.

[69] VMware. VIX API Documentation. `www.vmware.com/support/developer/vix-api/`.

[70] VMware. VMCI Sockets Documentation. `www.vmware.com/support/developer/vmci-sdk/`.

[71] VMware. vShield Endpoint. `vmware.com/products/vsphere/features-endpoint`.

[72] VMWare Inc. VMWare VMSafe security technology. `http://www.vmware.com/company/news/releases/vmsafe_vmworld.html`.

[73] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *SOSP*, 2005.

[74] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*, RAID'10, pages 158–177, Berlin, Heidelberg, 2010. Springer-Verlag.

[75] Wikibooks. QEMU/Monitor. `http://en.wikibooks.org/wiki/QEMU/Monitor`.

[76] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.