Efficient Hypervisor Based Malware Detection

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in
Electrical and Computer Engineering

Peter Friedrich Klemperer

B.S., Computer Engineering, University of Illinois at Urbana-Champaign
M.S., Electrical and Computer Engineering, University of Illinois at Urbana-Champaign

Carnegie Mellon University
Pittsburgh, PA

May 2015

Abstract

Recent years have seen an uptick in master boot record (MBR) based rootkits that load before the Windows operating system and subvert the operating system's own procedures. As such, MBR rootkits are difficult to counter with operating system-based antivirus software that runs at the same privilege-level as the rookits. Hypervisors operate at a higher privilege level than the guests they manage, creating a high-ground position in the host. This high-ground position can be exploited to perform security checks on the virtual machine guests where the checking software is isolated from guest-based viruses. The efficient introspection system described in this thesis targets existing virtualized systems to improve security with real-time, concurrent memory introspection capabilities.

Efficient introspection decouples memory introspection from virtual machine guest execution, establishes coherent and consistent memory views between the host and running guest, while maintaining normal guest operation. Existing introspection systems have provided one or two of these properties but not all three at once.

This thesis presents a new concurrent-computing approach – high-performance memory snapshotting – to accelerating hypervisor based introspection of virtual machine guest memory that combines all three elements to improve performance and security. Memory snapshots create a coherent and consistent memory view of the guest that can be shared with the independently running introspection application. Three memory snapshotting mechanisms are presented and evaluated for their impact on normal guest operation.

Existing introspection systems and security protection techniques that were previously dismissed as too slow are now be enabled by efficient introspection. This thesis explains why existing introspection systems are inadequate, describes how existing system performance can be improved, evaluates an efficient introspection prototype on both applications and microbenchmarks, and discusses two potential security applications that are enabled by efficient introspection. These applications point to efficient introspection's utility for supporting useful security applications.

Acknowledgments

My sincerest thanks to the members of my committee. Bryan D. Payne, Mahadev Satyanarayanan, Greg Ganger, and especially my adviser James C. Hoe. Thank you for the conference calls, markups, difficult questions, letters of reference, and countless feedback. James, thank you for your support in completeing this task we undertook together. You stood by me while I found my way through this process.

Thank you to the CUPS lab for taking me in and teaching me the ways of usability, especially Professors Lorrie Cranor and Lujo Bauer. I look forward to applying that knowledge throughout my career and for that I will be forever grateful.

Thank you to the members and staff of the Parallel Data Lab, but specifically to Joan Digney and Karen Lindenfelser, who always work so hard to make it all happen. The retreat provided a wonderful audience receptive to crazy new ideas and unparalleled networking opportunities.

I thank my fellow Hamerschlag Hall A-level labmates: Berkin Akin, Matthew Beckler (honorary), Adam Hartman (honorary), Eric Chung, Aaron Kane, Yoongu Kim, Brett Meyer (honorary), Peter Milder, John Porche (honarary), Eriko Nurvitadhi Marie Nguyen, Michael Papamichael, Malcolm Taylor, Yu Wang, Gabe Weisz, Guanglin Xu, and Milda Zizyte. The countless hours spent discussing projects over coffee were invaluable. Also thanks to my labmates in the CUPS lab, particulary Rebecca Hunt Balebako, Michelle Mazurek, Manya Sleeper, Rich Shay, Blase Ur, and Kami Vaniea. You all were my second home at CMU. Patrick, Brett, and Peter demonstrated how to be an excellent graduate student to me, even if I have not always lived up to their examples.

To my friends Casey Canfield and Michael Taylor, I will always remember our times together in Pittsburgh fondly. To the members of the Gigahurtz softball and hockey teams, we did not always win, but we definitely competed. To Brent Povis, Ryan Pocratsky, Jason Fox, and Shane Rice, thank

you for teaching me golf, the sport of Andrew Carnegie. All of you fine folks kept me relaxed and sane in the midst of the stress and trials these past six years.

To my parents Walter and Diane Klemperer who have supported me in so many ways. Thank you for the encouragement to undertake my Ph.D., for all the phone calls (please forgive me for all the phone calls I did not make also), the visits, and the wonderful holidays. I know I can always count on your love and support.

Finally, to my wonderful wife, Kristen, who began this journey with me and has given me so much love and support to follow it though. We make a really great team.

The research described in this thesis was made possible by the National Science Foundation via grant #DGE-0903659 and the Bertucci Graduate Fellowship.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis presents efficient introspection, a technique that supports the development of security applications in virtualized environments. Memory introspection is the process of examining virtual machine guest memory from the high-ground of the virtual machine host. The security application executes in a seperate environment, isolated by virtualization from the potentially infected guest.

Previously existing memory introspection techniques provided some, but not all, of the following three essential properties for efficient virtual machine introspection: coherent memory access, native introspection performance, and normal guest operation. Efficient introspection combines all three of these properties through high-performance memory snapshotting.

1.1 Motivation

In 2011 four new master boot record (MBR) based rootkits were released: TIDSERV.M, Fispboot, Alworo, and SMITNYL. Each of these rootkits infects the master boot record so that they will be loaded, with kernel privileges, before the Windows operating system (OS). Loading before the OS allows the virus to hide itself by subverting the kernel's own procedures. The TIDSERV.M-based Auleron botnet also uses its kernel level privileges to steal login, password and credit card data from the network traffic of infected systems. The threat posed by the Auleron botnet to banking data is difficult to counter with OS-based antivirus software. OS-based antivirus runs with the same privilege-level as the rootkits. The antivirus is vulnerable to tainting and spoofing by the rootkits, rendering detection much more difficult. Virtual machine (VM) based antivirus protects OSes from a high-ground position with a higher privilege than the viruses.

1.2 Memory Introspection

Memory introspection is a technique for hypervisor based security detection where the VM guest is inspected (or *introspected*) by the hypervisor. Existing introspection implementations typically mask memory access performance deficiencies with limited checking or long detection latencies. Slow memory access performance limits an introspection application's ability to check the entire memory state of a VM guest. To counter memory access performance limits, introspection application designers may try to seek out opportunities to investigate only smaller, more critical parts of the guest state so that the overall impact of slow memory access is minimized. Other works may periodically sample the guest memory, minimizing the memory access at any given sample, but over time build up a statistical knowledge of the overall guest state. While these contributions are certainly valuable, accelerating memory access capability directly enables new classes of introspection applications that were previously dismissed as too slow or resource intensive. Newly developed introspection applications will utilize the increased memory access support to provide real-time security protections against VM resident malware.

Inefficient introspection mechanisms not only limit introspection applications, but they can adversely effect guest performance. Guests that are taken off-line for checking or that have to compete for resources with inefficient introspection platforms may display poor performance. End-users may not tolerate poor performance and disable the introspection-based mitigation entirely, no matter the threats posed by rootkits.

1.3 Achieving Efficient Introspection

Increasing introspection performance requires answering two important research questions. First, how fast are the memory access performance requirements for a useful, real-time virtualization-based detection tool? Native memory performance is certainly a reasonable baseline but invites the second research question: can native speed be reached? How can the performance bottlenecks be understood, explained, and removed? This thesis answers these questions.

This thesis presents a concurrent-computing approach to accelerating hypervisor introspection of virtual machine guest memory. Existing virtual machine introspection tools are extended to provide parallelism by snapshotting guest memory. Snapshotting guest memory will allow the efficient introspection system to:

- decouple memory introspection from virtual machine guest execution,
- establish coherent and consistent memory views between the host-based introspection applications and the running guest,
- provide intelligent memory translation for native speed access to introspected memory spaces.

Existing introspection systems have provided one or two of these properties but not all three at once. This thesis will present a new real-time, concurrent-computing approach to accelerate hypervisor based introspection of virtual machine guest memory, which I call efficient introspection, that combines all three elements to improve performance and security.

The resulting system will add efficient introspection memory access capability while maintaining native guest performance. These efficiency increases will provide security system designers with greater flexibility to maintain guest performance and interactivity while increasing security checking capability. Efficiency is demonstrated through application and microbenchmark evaluation as well as through several potential introspection application investigations. Application benchmarks show that normal guest performance can be maintained under a variety of introspection scenarios. Microbenchmarking provides a systematic exploration of the introspection scenarios and helps explain why the application benchmarks perform well. The potential application discussion compares the efficient introspection to previously available introspection platforms and provides guidance for introspection application developers who are concerned with performance.

Detection techniques that had been formerly dismissed as too slow have been reevaluated in the context of efficient introspection and shown to be viable. This thesis explores two such techniques: memory-signature antivirus detection and network packet differential analysis. Specific limitations in introspection platforms had limited the utility of these application but they are now made possible with efficient introspection. These new techniques enabled by high performance memory introspection will increase protection for securing critical computing applications.

1.4 Thesis Contributions

The main contributions in this thesis are summarized as follows:

- Developing three requirements critical to the implementation of efficient introspection: coherent memory access, native memory introspection performance, and normal guest performance.
- Open-Source release of the efficient introspection prototype as an element of the LibVMI [1] introspection project.
- Evaluation of the performance impact of efficient introspection on application benchmarks reveals normal guest operation.
- Systematic exploration of introspection scenarios through microbenchmarking explains the behavior of the application benchmarks.
- Identification of key factors in efficient introspection application performance impact provide guidance for potential introspection application developers in the form of performance estimation techniques.
- Examination of potential applications of efficient introspection: memory-signature antivirus scanning and network packet differential analysis.

These steps towards defining and realizing efficient memory introspection leave several interesting research directions unaddressed. These open issues are summarized in Section 9.4.

1.5 Thesis Organization

Outline The rest of this prospectus is arranged as follows: Chapter 2 presents background on hypervisors, malware, and hypervisor based malware detection; Chapter 3 expands the concept of efficient introspection and how it will improve detection performance; Chapter 4 presents high-performance snapshotting mechanisms and describes their extension to LibVMI introspection platform; Chapter 5 discusses the evaluation of the efficient introspection prototype with application benchmarks;

Chapter 6 discusses the systematic microbenchmark evaluation of the efficient introspection prototype; Chapter 7 discusses potential security applications enabled by efficient introspection; Chapter 8 is related work; and Chapter 9 concludes the dissertation.

Chapter 2

Background

In this chapter I will provide background on hypervisors, discuss some previously existing platforms for guest memory introspection, and motivate the need for guest memory introspection through the discussion of a specific rootkit security threat called Mebroot.

2.1 Hypervisor Background

A hypervisor is a type of computer software that manages computing resources to allow multiple operating system instances, also known as guest virtual machines (VMs), to coexist on the same host physical computer. This thesis targets situations with existing hypervisor installations, regardless of whether those installations were chosen for reasons of consolidating hardware resources, expanding OS availability, or security. In this section I will provide some background on the implementation of hypervisors, mainly focusing on memory translation and isolation.

2.1.1 Memory Translation

Traditional memory translation supports multiple application running on the same computer by mapping each process into a unique contiguous virtual address space that references the more limited pool of physical memory present in the system. Pages from the virtual memory address space are translated into the physical memory address space in a manner specificied by the operating system. With virtual memory translation, the operating system can allocate resources between processes, prevent processes from interfering with one-another, and provide processes with predictable

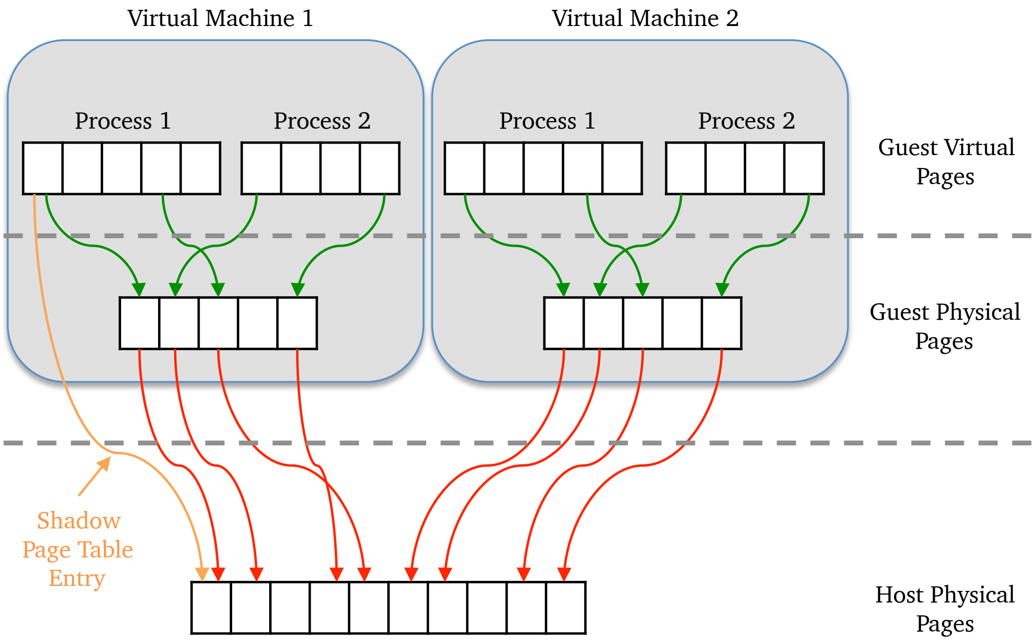


Figure 2.1: Depiction of two-level page mapping for two virtual machine guests, their processes on the same host, and shadow page table mappings. Figure borrowed with modification from VMware Performance Evaluation of Intel EPT Hardware Assist. http\protect\kern+.2222em\relax//www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf

memory address spaces.

Hypervisors provide a similar service as the operating system, but at a higher level of abstraction. Whereas the operating system facilitates the sharing of resources between multiple processes, the hypervisor facilitates the sharing of resources between multiple operating systems running on so-called virtual machine guests.

Memory translation is a key function of a hypervisor, allowing multiple guest instances to co-exist on the same physical computer. Virtualization adds another level of abstraction, the physical memory of the host (host physical memory) is addressed into the virtual machine guest physical memory (guest physical memory) address map which the guest uses to implement virtual memory (guest virtual memory). Each guest will behave as if it has control of the entire memory space and distribute that memory to it's processes, but the hypervisor must actively isolate each guest from one another so that the host resources can be shared between many guests.

Figure 2.1 illustrates several types of memory mapping. The normal virtual-to-physical memory mapping is demonstrated within each virtual machine in green. Host-physical-to-guest-physical memory mapping is demonstrated between the host and the virtual machines in red.

Shadow Page Tables

Shadow page tables are a specific implementation of the host physical memory, guest physical memory, and guest virtual memory address tranlsation hierarchy that can be implemented wihtout virtualization-specific X86 extensions. The guest operating system cannot be given direct control of the actual host physical memory to guest virtual memory address mapping. Instead, the host records a ”shadow page table” containing the guests intended physical-to-virtual memory mapping, as illustrated in Figure 2.1 with orange arrows. Whenever the guest attempts to modify it’s virtual memory mappings, the hypervisor intercedes and the shadow page table is then used to remap the host-physical-to-guest-virtual memory mappings as the guest operating system and it’s processes require. In this way, the shadow page table is hidden from the guest but maintains proper memory resource allocation ”from the shadows.”

X86 Virtualization

Just as the processor memory management unit (MMU) supports virtual memory translation for the operating system, new X86 virtualization extensions to the MMU support memory translation for the hypervisor. Intel VT-x and AMD-V are variations on the same theme: improving virtual machine performance through hardware supported memory address translation. These X86 virtualization replace the shadow page tables with a second level of address translation managed by the processor MMU, as illustrated by the red arrows in Figure 2.1.

Rather than having to trap to the hypervisor every time a virtual memory remapping occurs in the guest, the processor can use MMU and the second-level memory map to assure that the proper host-physical to guest virtual memory mapping is created. Hypervisor traps to handle guest page mappings had been a significant source of performance losses in virtualized systems before the implementation of the X86 Memory Virtualization extensions. Further, the need for shadow page tables has been eliminated, along with their corresponding memory storage overhead.

X86 Virtualization has not only improved memory access performance, but enabled several new technologies within the hypervisor like SamePage Merging [2]. SamePage Merging creates the capability to identify when identical host have been created on the host and merge them together through virtual memory mapping within or between guests. De-duplication of identical memory pages can lead to increased memory utilization efficiency.

2.1.2 KVM Hypervisor

The Kernel-based Virtual Machine (KVM) [3] is a leading open-source full virtualization platform that was originally authored by Kivity, Kamary, Laor, Lublin, and Liguori. The KVM project is in active development currently lead by Red Hat Software.

KVM requires a processor with X86 Virtualization capabilities. The KVM hypervisor is a kernel-space module that attempts to handle as many guest events as possible using the native virtualization capabilities of the CPU. Whenever the CPU cannot handle an event, the guest control is passed into an userspace emulator, typically the QEMU X86 processor emulator. QEMU handles events like initial set up of guest memory space, emulate I/O components like networking, and some video operations.

KVM is a capable hypervisor platform that can handle operations like guest pause-and-resume, guest migration between hosts, and automated guest storage management. Unlike the Commercial offerings from VMware, the source-code of KVM is freely available making KVM an attractive choice for research projects. Xen, another open-source virtualization platform, uses a custom microkernel for the host operating system, whereas KVM is hosted by a stock-linux kernel. The hosted nature of KVM and re-use of existing Linux kernel development knowledge were significant factors in the decision to develop the prototype presented this thesis as an extension to KVM.

2.2 Hypervisor Based Security

In this section I will first discuss some background on increasing security with hypervisor-based security and also how introspection performance limits can restrict security application development.

2.2.1 Hypervisor Based Introspection for Security

The memory protection of Hypervisors discussed in the previous section make them an attractive position for implementing security monitoring. The hypervisor runs at a high privilege level and has complete control of guest operation. The interface between a hypervisor and its guest is simpler and more slowly evolving than the interface between a program and an operating system and therefore creates a smaller attack surface. This smaller attack surfaces reduces the threat that the virus will escape the guest and directly subvert or disable the security monitoring system in the hypervisor.

Three common hypervisors were examined as a platform for this work: VMware ESX Server [4] is a bare-metal hypervisor sold by VMware, Xen by Barham et al [5] is a bare-metal hypervisor that runs the guests under its own custom host kernel, and KVM by Kivity et al. [3] is a hosted hypervisor that runs guests as Linux programs. I have chosen to implement my prototype using KVM because KVM is open-source, runs as a program within a standard Linux host so it can take advantage of standard Linux OS support, and is supported by an open source introspection library known as LibVMI [1].

2.2.2 Introspection Software: VMware VProbes

VMware VProbes is a debugging and introspection platform for the VMware hypervisor. VProbes scripts can instrument a running guest and have no cost when disabled. The intrumentation can provide many details about guest state such as memory contents, register state, and also insight into certain guest events like page-faults, interrupts, network-accesses and disk-accesses.

VProbes scripts are call-backs that are triggered whenever certain guest events occur. When the VMware hypervisor detects the event trigger, a handler calls the VProbes instrumentation, the instrumentation carries out it's task and saves it's results to a logging mechanism, the normal hypervisor event handler begins, and the guest continues.

More complex applications like `top` can be built by aggregating the results of samples. For example, the `pid` of the currently running guest process could be checked every time an interrupt is detected. The process that is observed to be running most often over a certain period of time could be inferred to be the top running process.

One primary goal of VProbes is to be safe, meaning not impacting a running guest performance. To enforce that safety, VProbe callback handlers cannot contain loops and have very limited stack size to prevent long running tasks. These callbacks execute in a short, finite period of time to avoid affecting guest performance. Some introspection mechanisms which require longer-term processing or more resources than are available to the VProbes intrumentation. In this case the introspection mechanisms must run in a seperate process from the VProbes and receive the results of the VProbes intrumentation through the VProbes logging mechanism. This process of passing guest state through the VProbes logging mechanism limits the scope of introspection capability.

2.2.3 Introspection Software: LibVMI

After encountering the disappointing performance related restrictions imposed by VMware VProbes, I sought out a more robust introspection platform. LibVMI is an expanded version of XenAccess, which was originally written by Payne, Carbone, and Lee [6]. The APIs provided by LibVMI support interacting with the virtual machine guest (pause/resume), inspecting guest memory, inspecting guest registers, and monitoring guest state. Various demonstrations are included with the software such as reading process lists from the guest memory, mapping symbol tables, and translating guest addresses. In addition, performance benchmarks are provided by LibVMI that measure various introspection behaviors such as translating virtual addresses, translating kernel symbols, and various memory access performance. These benchmarks will prove useful in demonstrating the efficiency and utility of the efficient introspection prototype. LibVMI is compatible with the Xen and KVM hypervisors. Since KVM is already supported by LibVMI, I can leverage the existing introspection technology and demonstrate efficiency improvements. While I have chosen KVM as the platform for this work, I do not foresee any limitations that would prevent applicability to other hypervisors like Xen or VMware ESX.

2.3 Detecting the Mebroot Rookit with Introspection

Rootkits are a class of malicious software that exists to provide priviledged access to a computer system while hiding that presence from detection by users or antivirus software. The Mebroot rootkit modifies the Windows operating system to hide it's presence on the disk and network traffic. This section will describe the modifications that Mebroot makes to the Network subsystems of the Windows operating system to hide itself from OS-based detection mechanisms and how the high-ground position of the hypervisor can be leveraged to detect Mebroot through introspection.

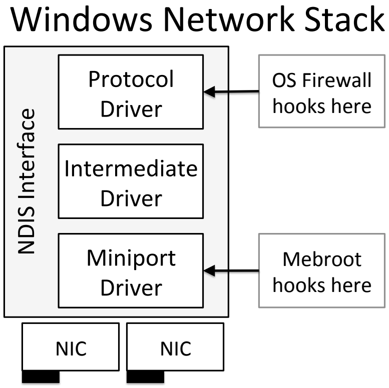


Figure 2.2: This block diagram describes the NDIS Network Stack found in Microsoft Windows operating system and where the network stack is hooked by the firewall and the mebroot virus.

2.3.1 Mebroot Threats

The Mebroot rootkit must send and receive network packets in order to receive control commands from it's operators and exfiltrate data found on the targets computer. Sending and receiving network packets without divulging it's presence to OS-based firewalls or packet monitoring software is accomplised by modifiying the Windows network stack.

Figure 2.2 illustrates the Windows NDIS network stack. The NDIS stack is an application programming interface designed to allow the development of hardware independent network drivers. At the top of the stack are protocol drivers that implement protocols like TCP/IP but also allows a convenient place for tools like firewalls and packet capture to examine the network traffic. Protocol drivers are also unique in the NDIS driver stack for their ability to communicate with user-applications directly. At the bottom of the stack, right above the actual hardware specific network interface card (NIC) drivers, are the miniport drivers. Miniport drivers control the packets accepted by a specific NIC and can be associated with any number of protocol drivers.

Typical Windows Firewalls hook the TCP/IP protocol driver in order to control incoming and outgoing traffic from various applications, as shown in Figure 2.2. User-applications are only able to communicate directly with the protocol drivers so this is an effective place to control application access to the network.

The GMER [7] rootkit detector team performed a reverse engineering analysis of the Mebroot rootkit and demonstrated that Mebroot creates it's own miniport driver in the NDIS network stack. The Mebroot miniport driver allows the rootkit to access the network directly while remaining hidden from the protocol level firewalls and packet capturing software.

2.3.2 Mebroot Virus Family

The Mebroot rootkit is part of a larger family of rootkits that are characterized by changing the master boot record of the target computer systems hard disk to gain control of the computer at the same privilege level as the operating system. A 2011 report by Hon Lau of Symantec Corporation [8] details past and emerging threats targetting the MBR.

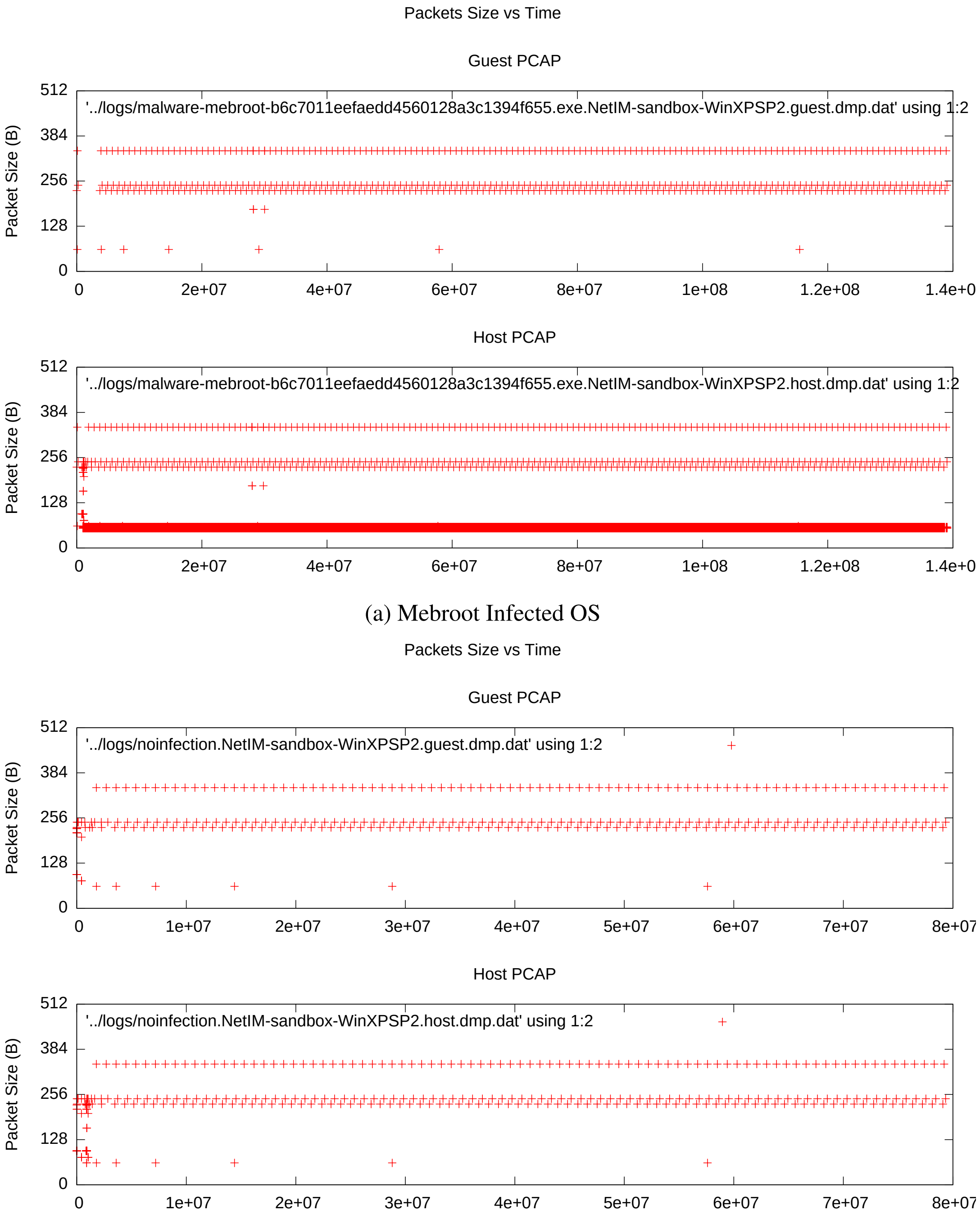
I have collected these threats in order to evaluate the effectiveness of the Network Integrity Manager against a broader classes of threats than just the Mebroot rootkit. Table 2.1 lists a subset of these threats that we were able to collect and evaluate, however, stone, mebratrix and bootlock were excluded. The Stone rootkit was deemed irrelevant as it is more of a rootkit development toolflow than a specific virus example. Samples of the Mebratrix virus have been obtained but could not be activated; confirming past experience with VMWare incompatability documented by Peter Kleissner [9]. The Bootlock virus was also excluded because it simply prevents boot of the infected system until a password is obtained.

2.3.3 Differential Mebroot Network Traffic Analysis

The Mebroot virus illustrates that phantom packets appear on the network but not on the guest operating system. Figure 2.3 illustrates this effect by showing four views of the packet traces over a period of approximately one day: 1) the guest packet trace of an infected guest, 2) the host packet trace of an infected guest, 3) the guest packet trace of an uninfected guest, and 4) the host packet

Threat	NetIM	Network Notes
No Infection	0	No traffic
Mebroot	180	Foreign DNS
TDSS4	0	Foreign DNS
Smitnyl	0	Foreign DNS
Fispboot	0	Foreign DNS
Alworo	0	Foreign DNS
Cidox	0	No traffic

Table 2.1: The available mebroot threats from the 2011 Symantec report with NetIM and DiskIM results and observation notes.



(a) Mebroot Infected OS

(b) Uninfected OS

Figure 2.3: Offline packet capture traces of (2.3(a)) an infected virtual machine guest and (2.3(b)) an uninfected virtual machine guest. For each virtual machine guest, a view from within the guest OS and from outside the guest OS, at the host, are presented. A large number of extra packets can be observed in the infected host PCAP trace, that are not observed in the infected guest PCAP trace or either of the uninfected traces.

trace of an uninfected guest. Each trace was the result of identical runs except for the infection of the guest OS with the mebroot virus. All network traffic observed in the traces was the result of the operating system as no applications were running at the time of the test except for the Network Integrity Manager guest probe and the virus, in the case of the infected trace.

The infected system showed interesting behavior approximately one hour hour after installation of the virus. The guest was observed to reboot itself and the suspicious behavior begins shortly upon startup. Comparing the guest packet traces of the infected and uninfected guests shows no significant differences. Two lines of traces at approximately 256 bytes and 350 bytes. These two lines also appear in the host-based trace observations of the infected and uninfected traces. The infected host-trace shows a third line of 50 byte packets that are not observed in the infected guest trace or either of the uninfected traces. These extra packets are observed by the host but not reported by the guest operating system, satisfying our condition for suspicious behavior.

Differential analysis comparing the network packets captured by the guest and host were compared to reveal more information about malicious communications. Table 2.2 presents a summary of both the host and guest communication with the target IP addresses and how many packets were sent or received. IP addresses associated with malicious behavior, wherein packets were observed by the host but not the guest, are highlighted in bold. Further analysis of the specific malicious IP addresses showed that the connections were attempting to connect with malicious DNS servers that had already been taken offline.

		Host			Guest	
Address	Total	Tx	Rx	Total	Tx	Rx
192.168.165.129	396	156	240	106	82	24
8.5.1.33	272	208	64			
192.168.165.255	58	0	58	58	0	58
192.168.165.254	48	24	24	48	24	24
98.124.193.1	6	3	3			
98.124.196.1	6	2	4			
74.125.113.147	4	2	2			
192.35.51.30	2	1	1			

Table 2.2: Network packet capture from both the uninfected host and Mebroot infected guest. Bold IP addresses indicate traffic only captured by the host. Further analysis indicated that the packets sent to the bolded IP addresses were DNS name resolution related.

Initial implementation involved maintaining a simple running count of host and guest events. An error is flagged if the difference between the host event count and guest event count is non-zero for longer than a certain time. This is troublesome for sustained events where there is a lag because the difference does not go to zero even when no hidden events occur.

The level of detail to make a match at is also important. Currently, events are simply matched on the basis of an event occurring. This has the advantage of being extremely cheap to compare and extremely cheap to gather traces from the host and guest. The imprecision of matching limits it's usefulness in identifying hidden packets. Increasing the precision of matching requires increased data collection. For the network, one could match, with increasing complexity, packet types (TCP,UDP,etc...), packet source/destination/port, or full packet matches. These decisions must be made carefully with regard to the impact of the performance on the data collection mechanisms in the host and guest, the bandwidth required to transmit the collected information from the host to the guest, and the overhead of performing the comparisons.

2.4 Background Summary

Hypervisor technology creates a high-ground position from which to observe the running VM guests. Hypervisor-based security exploits these unique properties of the hypervisor to detect secruity threats on the guest using the hypervisors position of privilege over the guest while isolated from a potentially malicious guest software. Rootkits, like Mebroot, evade detection from operating system-based virus mitigations by running with OS-level privileges and then subverting the OS-level mitigations. Hypervisor-based security restores privilege to the mitigation and creates advantage for rootkit detection. In the past poor performance has limited the application of security introspection but this thesis explores how increased efficiency can be found.

Chapter 3

Key Ingredients for Efficient Introspection

This chapter reintroduces memory introspection, develops three requirements for the realization of efficient introspection, clearly defines the requirements of efficient introspection, and, finally, compares existing platforms against the three requirements and finds them unsuitable.

3.1 Memory Introspection

Introspection is measuring virtual machine guest state from the hypervisor. Memory introspection has been an active security topic for many years but inefficient implementations have limited its utility for real-time applications. Work by Carbone et al. [10] has demonstrated large-scale kernel data verification but at the cost of requiring long analysis time due to limited memory access bandwidth and guest sequential access slowdown. Increasing the efficiency of memory introspection would enable kernel data verification and other similar memory bandwidth intensive techniques.

A second example of a high-memory use security technique is traditional signature-based antivirus. Signature-based antivirus involves calculating a checksum of each memory page on a computer and then comparing those checksums, or signatures, against a list of the checksums of memory pages containing known viruses. Signature-checking makes a better example for demonstrating the utility of efficient memory introspection than kernel verification because signature checking scales linearly with the size of the memory being checked whereas kernel verification depends on the state of the specific running kernel. Currently, signature-based antivirus checking can be implemented from the hypervisor but performance inefficiencies require tradeoffs in guest performance impact against the amount of time taken to completely scan memory: either, (1) scan memory quickly but impact guest performance, or, (2) maintain guest performance but take a long time to scan. In the next two sections I will discuss why neither of these outcomes are acceptable.

3.2 Developing Requirements for Efficient Introspection

The two motivating examples discussed in the previous section, show how inefficient introspection systems of the past limited the scope of introspection application development. In this section I will develop requirements for efficient introspection that will enable the developement of introspection applications that were previously dismissed for implementation with inefficient introspection platforms.

3.2.1 Pausing is too slow so we need coherency

One simple method for efficiently implementing coherent memory introspection is to pause the guest, quickly perform a check, then allow the guest to continue. If the check is performed quickly enough then the performance penalty to the guest may be acceptable. As checks require more time to complete or need to be performed more frequently, then the performance impact will increase, possibly reaching unacceptable levels.

Small checks, like rebuilding a process list, will have a relatively small runtime and can be performed with variable frequency. The Lycosid system by Jones et al. [11] provides an important example. Lycosid reveals hidden processes using a statistical method to compare the reported process list with a list of observed processor states. Increasing the frequency that the processor states are observed has a cost in guest performance but increases the statistical likelihood that a hidden process will be discovered.

As long as guest execution and checking are linked, larger checks like signature-checking the entire memory space will require long pauses with unavoidable performance impact. Decoupling guest execution from checking can be achieved by exploiting parallelism through multi-threaded execution but will require careful implementation to maintain secure and predictable behavior.

3.2.2 Parallelism without coherency is insufficient

A simplistic method of decoupling guest execution from checking is to simply perform the checks as needed on the running guest. This method allows long checks to be carried out over a longer period of time with less impact on the guest running time but creates several problems. Primarily, reading memory state from a running process can produce inconsistent and incoherent results. In the case of validating kernel memory state, as in the work by Carbone et al. [10] discussed previously, if we don't guarantee memory state is unchanging while rebuilding a process list then we may get a broken list if we were to scan the list as a process is being removed. Further, polymorphic viruses, like those described by Ször and Ferrie [12], encrypt themselves using self-modifying code techniques to hide from signature based antivirus mechanisms and only decrypt themselves while performing critical (malicious) operations that might be missed if coherency were not maintained. Finally, precisely timing the checks to coincide with system events becomes very difficult on a running system. For these reasons, parallelizing guest execution and checking without regard for coherence will increase performance but at the cost of increased checking complexity and probable security vulnerabilities; parallelism without coherency is insufficient for improving memory introspection performance for security applications.

3.2.3 Efficient Introspection: Parallelism with Coherency

This thesis decouples guest execution from checking in a coherent manner through an approach that I call efficient introspection. In efficient introspection the introspection application programmer specifies a moment in time for the check to begin and the underlying platform creates a lightweight snapshot of the guest state at that moment for the introspection application to access. The guest then continues operation in parallel with the introspection application. Upon completion of the check, the snapshot is no longer required so it is destroyed. The scope of the snapshot can also be specified by the introspection application programmer according to each application's needs. As shown later in this chapter, existing hypervisor introspection technology is insufficient to support efficient introspection.

3.3 Requirements for Efficient Introspection

Three requirements were developed for efficient introspection: first, native memory performance for introspection; second, coherent memory views for introspection. and, third, normal guest performance. This section will further define these requirements.

3.3.1 Requirement 1: Native Memory Introspection Performance

Native memory performance is defined as the capability to introspect guest memory with the same performance as the host can introspect it's own memory. Evaluating whether native memory performance is achieved in a specific introspection implementation is relatively straight-forward. Memory performance microbenchmarks, like *lmbench* [13], will be discussed later in the thesis and can provide access performance for the host that can be compared with the performance of the guest introspection interface.

Slow memory introspection interfaces or memory translation bottlenecks will limit the scope of introspection applications possible with the platform. Large tasks like kernel data structure verification that require traversing large swaths of memory would take long periods of time with slow memory introspection interfaces. Native memory performance ensures that the broadest possible classes of introspection applications can be supported by efficient introspection.

3.3.2 Requirement 2: Coherent Memory Views

In this thesis coherence is defined as the guest-view of the virtual machine state matching the host-view of the virtual machine state at the same moment in time. Requirement 2 — coherent memory views — specifically refers to the guest memory view matching the introspected memory state at a single moment in time.

As discussed earlier, if the introspection mechanism is looking at old or stale guest state, critical detection information could be lost. In the absence of coherent memory views, introspection applications would have to maintain coherence themselves or else suffer without it. Implementing a coherence mechanism is a high-bar for introspection application developers and is more appropriately implemented in the hypervisor. Building coherent memory views into efficient introspection reduces the overhead on introspection application developers and the possibility for errors.

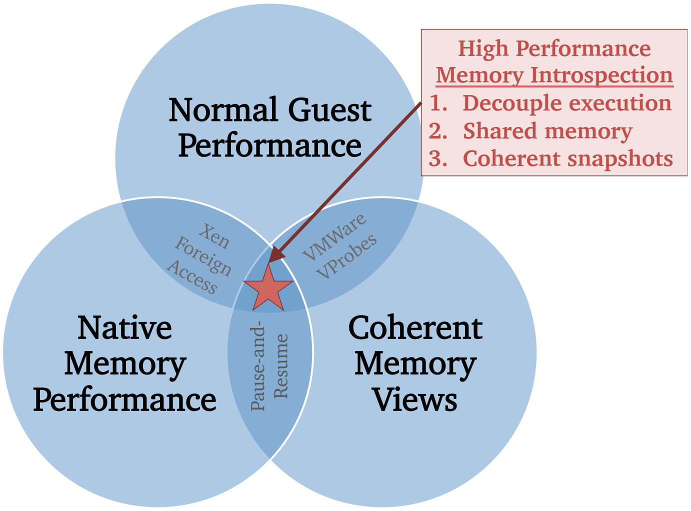


Figure 3.1: Three capabilities are required to support efficient introspection: normal guest performance, memory introspection at native access speeds, and coherent views of the guest memory from the host. Existing introspection platforms like xen-foreign-access, VMware VProbes, and Pause-and-Resume LibVMI, only support two requirements of the three. Only efficient introspection supports all three requirements.

3.3.3 Requirement 3: Normal Guest Performance

Normal guest performance must be defined on a per-case basis, but generally means that, to an external observed, the guest behaves the same with efficient introspection as without. Performance can be measured using whatever metrics are relevant for that specific application or situation. A web server might be measured in terms of http connections supported per minute. A machine learning application might be measured in terms of runtime. The key element here is that end-users will not reject the efficient introspection platform entirely for having an adverse impact on the task that they actually want to complete.

3.4 Existing Introspection Platforms Inadequate

The introspection mechanisms provided by the current major virtualization platforms – VMware VProbes and the LibVMI interfaces to Xen and KVM – are insufficient for the requirements of efficient efficient introspection.

- VMware VProbes is an introspection mechanism supported by VMware Workstation and ESX. VProbes offers low-overhead introspection primarily targeted at counting system events.

Even page-scale memory introspection capabilities are not offered, which limits VMware VProbes utility for larger memory introspection techniques like signature checking. Coherence is maintained but VMware controls the run length of a given probe to prevent performance degradation which limits VProbes' utility as a general purpose tool.

- Xen offers native performance zero-copy memory sharing through it's XenControl API. While memory access is very fast, pausing the VM is the only way to ensure consistent and coherent introspection. As discussed earlier, pausing the guest incurs significant overhead under many useful introspection applications.
- KVM exposes guest memory through either a virtual serial interface or full memory dumps to disk. A set of experimental patches have been produced by the authors of LibVMI to expose page level access, but the memory is copied out page-by-page, limiting performance [14].

An efficient implementation of efficient introspection will require fast zero-copy memory sharing like that found in Xen combined with a memory management scheme to ensure consistent and coherent memory views. Figure 3.1 illustrates the three requirements for efficient introspection, the limitations of existing platforms in meeting those requirements, and how efficient introspection satisfies all three.

Increased introspection performance over previous techniques will be accomplished in two ways: first, through decoupling guest execution from the introspection execution and, second, through creating high-performance, coherent memory sharing. Current memory sharing approaches are insufficient for implementing efficient efficient introspection. In order to move forward and increase efficiency, new mechanisms will have to be developed which combine the fast zero-copy sharing approach offered by XenControl with smart memory management to ensure efficient introspection through an efficient snapshotting mechanism. Looking at other common techniques–like migration of a VM between hosts over a network–will inform the development of new snapshotting mechanisms.

3.5 Summary

This section develops and then clearly defines three requirements for efficient introspection: native memory performance, coherent memory views, and normal guest performance. These three requirements were not met by existing introspection platforms from VMware and the LibVMI project. The next chapter will introduce high-performance snapshotting as the key detail for satisfying all three requirements of Efficient Introspection.

Chapter 4

Implementing Efficient Introspection by Snapshotting

The previous chapter developed three requirements for efficient introspection and put forward high performance memory snapshotting as a practical solution that satisfies all three requirements.

This chapter presents three specific memory snapshotting mechanisms, provides guidance applying the snapshotting mechanisms to different computing scenarios, presents the specific implementation details of the efficient introspection high-performance snapshotting in the KVM hypervisor, and describes integration of the snapshotting with the LibVMI introspection platform.

4.1 High Performance Snapshotting

The efficient introspection prototype supports the creation and management of memory snapshots that are made available to introspection applications through a shared memory interface. The actual implementation of the prototype consists of modifications to the KVM virtualization platform and the LibVMI introspection platform. The block diagram in Figure 4.1 illustrates how the shared memory interface promotes efficient introspection between the Introspection Application and the VM Guest. In this section I will discuss the details of these modifications.

Snapshotting guest memory is key to providing coherent memory views to introspection applications. In order to assure that the snapshot faithfully represents the state of the guest memory at a single point in time, the guest is paused at that point in time, the guest memory is copied into

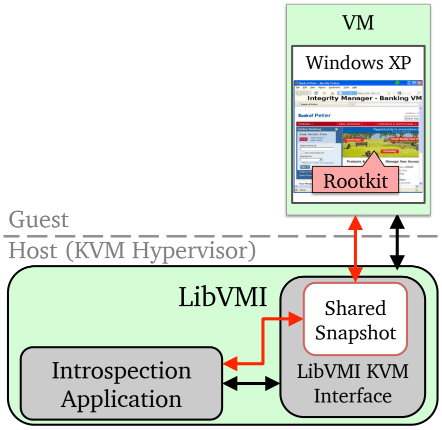


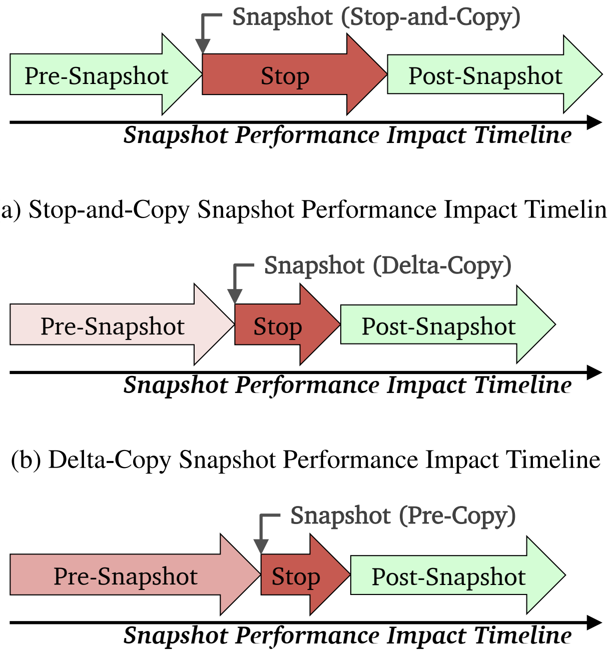
Figure 4.1: This block diagram illustrates how the shared snapshot interface is provides the Introspection Application with a view into the memory of the VM Guest.

the snapshot using KVM built-in memory access mechanisms, and then the guest is restarted. During the time where the guest is snapshotting (pausing, copying, and restarting) the guest cannot make forward progress. In order to minimize snapshot overhead and meet the first criteria for efficient introspection, normal guest operation, several mechanisms were developed for snapshotting memory. The snapshotting mechanisms have been named stop-and-copy, delta-copy, and pre-copy. Figure 4.2 shows the performance impact of various snapshot implementation mechanisms, which are discussed below.

4.1.1 Stop-and-Copy Snapshot

The Stop-and-Copy snapshot is the simplest of the three mechanisms. In Stop-and-Copy, the guest is simply stopped (paused), the guest memory is copied out page-by-page into the snapshot, and then the guest is restarted. The snapshot memory must be the same size as the guest memory. Standard POSIX SHM shared memory objects manage shared access to the snapshot memory for both the hypervisor and introspection application processes. The hypervisor must have write access to the snapshot memory but the introspection application is only provided read access. The relative simplicity of the Stop-and-Copy snapshotting mechanism lead its choice for the initial implementation of high-performance snapshotting.

Stop-and-Copy snapshotting has several benefits and drawbacks. Snapshot stop-time is independent of guest load since every byte of guest memory must be copied for every snapshot. This



(a) Stop-and-Copy Snapshot Performance Impact Timeline

(b) Delta-Copy Snapshot Performance Impact Timeline

(c) Pre-Copy Snapshot Performance Impact Timeline

Figure 4.2: Snapshot performance timelines for Stop-and-Copy 4.2(a), Delta-Copy 4.2(b), Pre-Copy 4.2(c). Worse performance is indicated as darker red and no-impact is indicated in green.

property is particularly important in security contexts where malicious guests might attempt to influence security mechanism behavior. Another advantage of the Stop-and-Copy snapshotting mechanism is that it has no active mechanisms during the run-time of the guest. As a result, unlike other mechanisms, Stop-and-Copy snapshotting will not influence guest performance when not actively snapshotting. Simplicity of implementation is a major advantage for Stop-and-Copy snapshotting. The significant drawback is that stop-and-copy is very slow as each byte of guest memory must be copied to complete each snapshot. Figure 4.2(a) illustrates the performance impact of stop-and-copy snapshotting and highlights how stop-and-copy only impacts guest performance during a snapshot.

4.1.2 Delta-Copy Snapshot

The delta-copy mechanism tracks write to memory in the guest and only copies pages that have changed since the previous snapshot. The implementation of delta-copy snapshotting in the KVM prototype leverages the existing dirty-page tracking mechanisms built into KVM for other virtual machine management functionality. The guest-snapshot is performed as a stop-and-copy snapshot except that before the the guest is restarted, all the guest memory page state is marked as "clean." After the guest has restarted, as the guest writes to memory, the KVM dirty-page tracking mechanism maintains a list of those written "dirty" pages. When the next, and all subsequent, snapshots are taken, only the "dirty" pages will have changed from the previous snapshot, so only those pages will have to be copied into the snapshot and then the list of dirty pages is cleared. The snapshot memory must be the same size as the guest memory. Standard POSIX SHM shared memory objects manage shared access to the snapshot memory for both the hypervisor and introspection application processes. The hypervisor must have write access to the snapshot memory but the introspection application is only provided read access. Figure 4.2(b) illustrates the performance impact of delta-copy snapshotting and highlights how delta-copy impacts guest performance during a snapshot and only minimally impacts the guest before the snapshot.

The delta-copy snapshotting mechanism has several advantages and disadvantages. A major benefit of delta-copy snapshotting is that snapshotting stop times are reduced significantly for guest-loads that do not write many guest pages. Only newly written pages are copied into the snapshot, saving the cost of overwriting pages that had not changed and were already stored in the snapshot. This benefit is multiplied when snapshot frequencies increase because the guest has less time to write pages between snapshots. As we will see in the Application Benchmarking chapter, many interesting guest loads write a relatively small subset of the available memory, or write to memory infrequently, allowing significant performance increases over stop-copy to be realized. Delta-copy snapshotting has several drawbacks. Foremost, especially for security applications, is that the snapshotting time is dependent on the specific guest load memory writing pattern. A malicious guest could attempt to overload the snapshotting mechanism by creating artificially large dirty page sets. A malicious guest application could accomplish this by writing one byte to each page in a large memory allocation. Fortunately, the copying overhead is capped at the size of the guest snapshot, essentially the same overhead as stop-and-copy. A second drawback is that the dirty page tracking mechanism must be enabled during guest operation, potentially creating interactions and side effects on guest behavior. In the case of the KVM specific implementation, efficient dirty-page tracking is an established mechanism already present in the hypervisor, so the impact is barely measurable. Other implementations of delta-copy snapshotting in other hypervisors will have to evaluate the performance overhead of dirty page tracking on the guest performance, but dirty-page tracking is very common in all hypervisors as it is used to support page table management and guest migration.

4.1.3 Pre-Copy Snapshot

The pre-copy mechanism starts with the delta-copy mechanism but adds a provision for eagerly *pre-copying* pages into the snapshot ahead of snapshot stop time, thereby reducing the number of pages copied during the snapshot stop time. Just as with delta-copy, after the guest memory has been copied into the snapshot the guest dirty page list is cleared before restarting the guest. The introspection application can then use snapshot, but unlike the previous two mechanisms, the introspection application can *release* the snapshot back to the hypervisor. After the introspection application has released the snapshot it should not read from the snapshot as the snapshot memory state is undefined. After the hypervisor receives word that the introspection application has released the snapshot, the hypervisor can spawn a Pre-Copy thread, that periodically scans the dirty page list, marks the page as clean, and then pre-copies the dirty memory page from the guest into the snapshot. In this way infrequently written pages can be written into the snapshot while the guest is still running, reducing the number of dirty pages that have to be copied during the snapshot, and reducing snapshot stop time. Figure 4.2(c) illustrates the performance impact of pre-copy snapshotting and highlights how pre-copy impacts guest performance during a snapshot and but may substantially impact the guest before the snapshot while the pre-copy mechanism competes with the guest for bandwidth.

The Pre-Copy snapshotting mechanism has several advantages and disadvantages but they are less clear-cut and will have to be evaluated on a case-by-case basis. The major benefit of pre-copy is that dirty pages that have been successfully pre-copied before the snapshot will not have to be copied during snapshot stop time, reducing snapshot stop time and improving guest load performance. The drawbacks of pre-copy snapshotting is that the pre-copy mechanism competes with the guest for memory access. Each pre-copied page reduces bandwidth available for the guest. Introspection applications must release the snapshot back to the pre-copy mechanism, potentailly reducing time available for completeing introspection. The process of synchronizing the dirty page list can be expensive, specifically KVM's implementation of the dirty-page tracking, which is not a problem when done once at snapshot time, like in delta-copy snapshotting, but can adversly impact normal guest performance if done repeatedly by the pre-copy thread. Finally, the advantages of pre-copy, shorter snapshot stop times, are ameliorated by increasing the time between snapshots. The pre-copy mechanisms require time between snapshots to perform their task, so while longer time available for pre-copying pages yields shorter snapshot stop times, those longer times between snapshots prevent the performance gains from being realized. Tuning the time between snapshots and the rate of pre-copy to the needs of each introspection scenario may be tricky.

4.1.4 Snapshotting Mechanism Guidance

Each of the snapshotting mechanisms – stop-and-copy, delta-copy, pre-copy – will affect guest performance in different ways. Particularly interesting are delta-copy and pre-copy mechanisms performance impacts that are dependent upon the guest load.

Stop-and-copy snapshotting performs well in scenarios that combine a large working set and large memory bandwidth requirements where delta-copy or especially pre-copy bookkeeping overheads would reduce performance but not reduce stop-time overheads.

Delta-copy snapshotting performs well in scenarios combine small working sets that can be quickly copies with frequent memory use that reduces the effectiveness of the pre-copy mechanism to further reduce the working set.

Pre-copy snapshotting combines large working sets with infrequent uses. Typically large working sets are not optimal for delta-copy mechanisms but infrequent use makes pre-copy effective. Further, the time between snapshots must be significantly long to allow for the introspection application to release the snapshot and then for the pre-copy mechanism has eagerly copy a significant number of dirty pages. Further, the introspection application must be amenable to releasing the snapshot. These requirements conspire to reduce the benefit of pre-copy through amortization of more expensive snapshotting mechanisms over time with infrequent snapshotting.

4.2 KVM/QEMU Hypervisor Modifications

The KVM/QEMU hypervisor that was modified to implement the each of the three snapshotting mechanisms outlined in the previous section as well as share the snapshots over a POSIX shared memory interface. Both parts of the KVM/QEMU hypervisor, a kernel module known as KVM and user-space emulator known as QEMU, were used to implement the efficient introspection prototype.

4.2.1 KVM Host Linux Kernel Module

KVM is the open-source, host-based full virtualization solution was modified to support efficient introspection. Currently, the KVM kernel module provides support to the hypervisor for live guest migration between hosts over a network. The live migration facilities rely on dirty-page marking features provided by the kernel module to manage memory coherency between the source and target hosts. These page marking facilities form the basis of the efficient introspection delta-copy and pre-copy snapshotting mechanisms.

4.2.2 QEMU Modification Details

QEMU is a generic and open source machine emulator and virtualizer [15]. The KVM Linux kernel module requires QEMU to provide userspace virtualization support. Extensions to the QEMU Monitor Protocol (QMP) will support snapshotting operations between the introspection library and the hypervisor are listed in Listing 4.1 and described below.

KVM Snapshot Create

The `snapshot-create` command causes the hypervisor to initiate a snapshot of specified size. Two versions of this function were written, one that implements the stop-and-copy snapshot mechanism and a second version that implements delta-copy and pre-copy.

KVM Snapshot Destroy

The `snapshot-destroy` command allows the introspection application to release control of the shared memroy snapshot for the purposes of freeing the shared memory snapshot. Ordinarily, this function is only invoked at the completion of the introspection appplication.

KVM Snapshot Release

The `snapshot-release` command allows the introspection application to release control of shared memory snapshot back to the hypervisor for the purpose of initiating pre-copy. The hypervisor can then spawn the memory pre-copy thread that will copy pages into the snapshot at the appropriate rate. The introspection application must release the snapshot before each snapshot for

Listing 4.1: KVM QMP Command Extensions for efficient introspection

```
##
# @snapshot−create
#
# Create a memory snapshot with POSIX shared memory.
#
# @filename: store at /dev/shm/filename
#
# Returns: json−int the size of the memory snapshot in bytes.
#
# Since: 1.6
##
{’command’: ’snapshot−create’, ’data’: { ’filename’: ’str’ },
  ’returns’: ’int’ }

##
# @snapshot−destroy
#
# Destroy the memory snapshot with POSIX shared memory.
#
# @filename: Destroy snapshot stored at /dev/shm/filename
#
# Returns: none.
#
# Since: 1.6
##
{’command’: ’snapshot−destroy’, ’data’: { ’filename’: ’str’ } }

##
# @snapshot−release
#
# Release the memory snapshot (does not destroy the snapshot)
# Note:
# Releasing the snapshot allows the pre−copy mechanism to
# update the POSIX shared memory in an attempt to reduce
# snapshot stop time. The POSIX shared memory will be
# in an undefined state until the snapshot−create command
# is run again.
#
# Parameters: none
#
# Returns: none
#
# Since: 1.6
##
{’command’: ’snapshot−release’ }
```

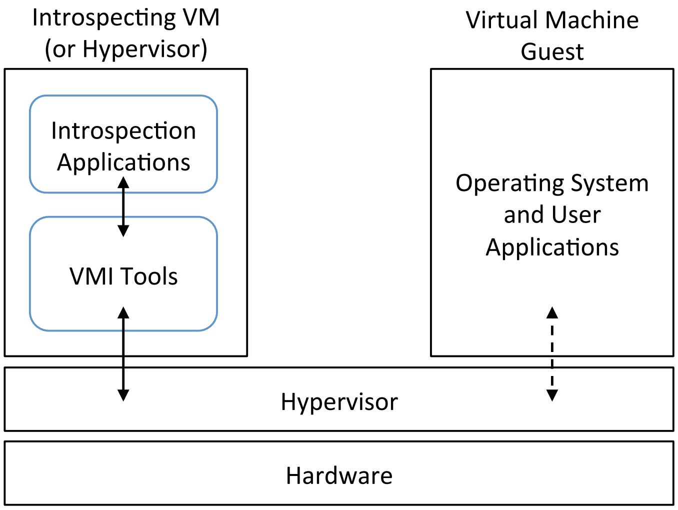


Figure 4.3: VMI Tools operation block diagram showing Introspection VM, guest being introspected upon, hypervisor, and hardware. Figure borrowed with modification from VMI Tools website. http://code.google.com/p/vmitools/

it to gain the benefit of the pre-copy snapshot mechanism. If the snapshot is not released, then a normal delta-copy snapshot is performed.

Other KVM Commands

Several more utility QMP commands were added. The `Pre-Copy-Xfer-Limit` command sets the maximum pre-copy transfer rate or allow it to be unlimited. The `Dirty-Page-Count` command returns the current dirty page count without snapshotting the guest and was used for testing purposes.

4.3 The LibVMI Project Modifications

As discussed earlier in Section 2.2.1, LibVMI is a set of tools that enable virtual machine introspection that have been developed by Bryan D. Payne. It should be noted however that LibVMI was not used to implement the testing frameworks used in the evaluation chapters of this thesis because I wanted to isolate the effects of the snapshotting mechanism from those of the LibVMI Library or the introspection applications.

LibVMI is implemented as a set of tools, operating in the hypervisor or introspection virtual machine guest, that interacts with the hypervisor to monitor the guest being introspected upon; as shown in the block diagram in Figure 4.3. The LibVMI introspection library is designed to use a pause/resume coherency model but can be modified to suit more efficient efficient introspection implementations. In fact, in efficient introspection supported platforms, the LibVMI pause and resume functions built into existing introspection applications can be remapped to create a snapshot (pause) and destroy the snapshot (resume) with minimal introspection application modification. LibVMI is a modular introspection system supporting multiple virtualization platforms like KVM and Xen.

The modifications required for LibVMI to support efficient introspection have been made as an additional module alongside the KVM and Xen platforms. I would like to recognize the contribution of summer intern Guanglin Xu performed the hard work of implementing my proposed API changes and releasing them to the open-source community.

4.3.1 LibVMI API Changes

The LibVMI modifications support managing snapshots, but also efficient memory access and guest address translation. The LibVMI API modifications required for efficient introspection are in Listing 4.2. and are summarized below.

LibVMI Initialize

The `vmi_init` function had to be modified accept a flag indicating that a shared-memory KVM snapshot module should be used instead of the previously existing Xen and KVM modules. A snapshot is taken at initialization to confirm the type of guest and perform other housekeeping tasks that require identification of the guest.

LibVMI SHM Snapshot

The `vmi_shm_snapshot_create` function sends a QMP `snapshot-create` command to the hypervisor, opens the newly created snapshot, and prepares LibVMI to serve requests for pointers into the guest snapshot before returning control to the introspected application.

Listing 4.2: LibVMI Project API Extensions for efficient introspection.

```

// vmi_init creates a new vmi instance
// added new flag VMI_INIT_SHM_SNAPSHOT
// to indicate that a snapshot should be taken.
status_t vmi_init(
    vmi_instance_t &vmi,
    uint32_t flags,
    char *name);

// vmi_shm_snapshot_create snapshots the
// virtual machine under introspection.
status_t vmi_shm_snapshot_create(vmi_instance_t vmi);

// vmi_shm_snapshot_destroy destroys the
// snapshot of the virtual machine under
// introspection.
status_t vmi_shm_snapshot_destroy(vmi_instance_t vmi);

// vmi_get_dgpma returns a pointer to a buffer
// containing the snapshot of the physical memory
// for the virtual machine under introspection
// of count bytes at the specified address.
size_t vmi_get_dgpma(
    vmi_instance_t vmi,
    addr_t physical_address,
    void **buf_ptr,
    size_t count);

// vmi_get_dgpma returns a pointer to a buffer
// containing the snapshot of the virtual memory
// for the virtual machine under introspection
// of count bytes at the speicifed address
// in pid process.
size_t vmi_get_dgvma(
    vmi_instance_t vmi,
    addr_t virtual_address,
    pid_t pid,
    void **buf_ptr,
    size_t count);

// vmi_shm_snapshot_release releases the snapshot
// of the virtual machine under introspection.
// The snapshot contents is undefined until
// until vmi_shm_snapshot_create is called again.
status_t vmi_shm_snapshot_release(vmi_instance_t vmi);
```

LibVMI SHM Destroy Snapshot

The `vmi_shm_snapshot_destroy` function closes the shared memory snapshot and sends a QMP `snapshot-destroy` command to the hypervisor. This function is typically only called at the completion of an introspection application.

LibVMI Get Physical Address

The `vmi_get_dgpma` function returns a pointer to a buffer containing the guest physical memory of the guest at the specified address. This function is not available without snapshotting support.

LibVMI Get Virtual Address

The `vmi_get_dgpma` function returns a pointer to a buffer containing the guest virtual memory of the guest for the specified address and process. This function is not available without snapshotting support.

LibVMI SHM Release Snapshot

The `vmi_shm_snapshot_release` function sends a QMP `snapshot-release` command to the hypervisor, allowing the pre-copy snapshot mechanism to precopy memory until `vmi_shm_snapshot_create` is called again.

4.4 Example Minimal LibVMI Application

Now that I have outlined the proposed prototype, I would like to describe a simple introspection program that utilizes efficient introspection.

The code example in Listing 4.3 demonstrates how introspection snapshots the guest, finds the address of the main system process, reads the memory of the main system process, and destroys the snapshot. This simple example illustrates the basic features of introspection. Efficient introspection enabled the application to use a more efficient `memcpy` function to directly copy the memory using the pointer into guest memory. Previously, without efficient introspection, guest memory was read iteratively through port style `read` functions.

Listing 4.3: VMI Tools program example source code with modifications for improving introspection performance with efficient introspection.

```
#include ”libvmi/libvmi.h”
#include ”common.h”

int main() {
        vmi_instance_t vmi;
        addr_t start_address;

        int buf_size = 256;
        unsigned char *buf = malloc(buf_size);

        /* initialize the xen access library */
        vmi_init(&vmi, VMI_AUTO | VMI_INIT_COMPLETE, ”GuestVMName”);

        /* snapshot replaces pause call (vmi_pause_vm) */
        guest_snapshot_ptr = vmi_snapshot_create(vmi);

        /* find address to work from */
        /* get virtual address from kernel symbol table for symbol PsInitialSystemProcess */
        start_address = vmi_translate_ksym2v(vmi, ”PsInitialSystemProcess”);
        /* translate virtual address to physical address for introspection */
        start_address = vmi_translate_kv2p(vmi, start_address);
        /* address translations are cached to improve performance */

        /* read location of PSInitialSystemProcess physical address in guest memory */
        /* previously vmi_read_pa functions were required but now kernel driver
           enables direct shared memory access */
        memcpy( guest_snapshot_ptr[start,address], buf, len(buf) );

        /* throw away snapshot instead of resume (vmi_resume_vm) */
        vmi_snapshot_destroy( guest_snapshot_ptr );
        vmi_destroy(vmi);
        return 0;
}
```

Chapter 5

Application Benchmark Evaluation

This chapter demonstrates that high-performance snapshotting can provide normal guest operation for a battery of introspection scenarios with application benchmarks as guest loads. The next chapter will use microbenchmarks to systematically explore the introspection scenarios, explain why and how high-performance snapshotting is successful.

5.1 Benchmark Testing Procedure

Before discussing the application benchmarks, I will describe the procedure that was developed for testing the performance of the application benchmarks. Figure 5.1 illustrates the application benchmark testing procedure. The test begins when the application benchmark is started in the VM guest, then the guest memory is snapshotted periodically. The application benchmarks are run to completion and the result of the benchmark (runtime, bandwidth, requests/second, etc.) is recorded. The test fails if the snapshotting cannot be completed in the period specified between snapshots. The host-guest shared memory is read between snapshots to mimic the behavior of introspection. The test fails if the specified introspection read cannot complete before the next scheduled snapshot.

Each of the Application Benchmarks was tested in several introspection scenario configurations to present a range of results. The snapshot periods – the time between snapshots – were varied between one and three-hundred seconds. This range was chosen because most stop-copy snapshots could not complete more frequently than once second even on an unloaded guest. Three-hundred seconds was long enough that the Application Benchmarks could complete between snapshots (ex-

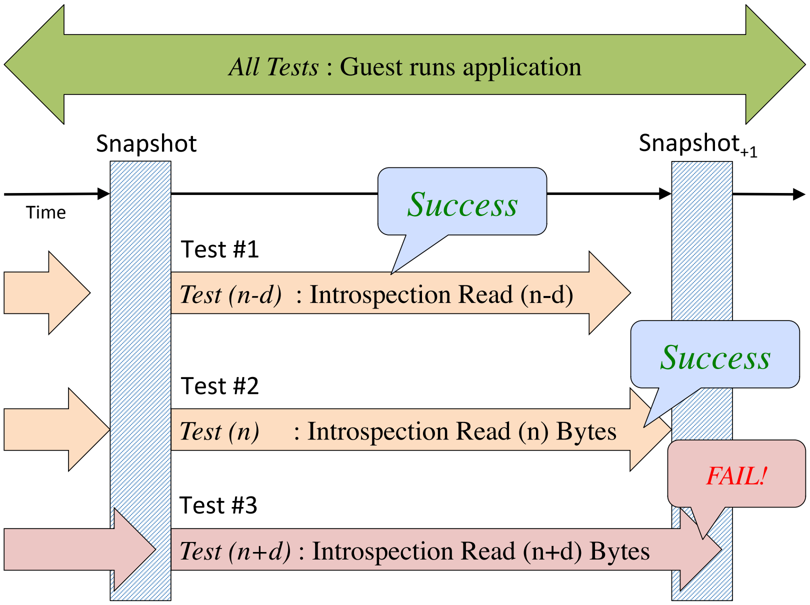

All Tests : Guest runs application
Snapshot
Snapshot$_{+1}$
Time
Success
Test #1
Test (n-d) : Introspection Read (n-d)
Success
Test #2
Test (n) : Introspection Read (n) Bytes
FAIL!
Test #3
Test (n+d) : Introspection Read (n+d) Bytes

Figure 5.1: Block diagram describing the application benchmark testing procedure. In Tests #1 and #2 introspection completes successfully before the next snapshot period begins. Test #3 fails because introspection could not complete before the scheduled start of the next snapshot period.

cept for the Kernel Build, described later). The guest VM was configured with two virtual CPUs and 2048 GB of memory. The snapshots were configured for 2048 GB (the full guest memory space) in both the stop-copy and delta-copy configurations. Introspection varied between 0 and 5000 MB read from the shared memory between each snapshot event.

5.2 Application Benchmarks

Five benchmarks were chosen as representative applications for evaluating the impact of efficient introspection on normal guest operation. These benchmarks are as follows: Kernel Build, that consists of building the linux kernel; ClamAV Antivirus Scan, an antivirus scanner; Apache Web Server, a webserver; Netperf Network Performance, a network performance benchmark; and, Weka Machine Learning, a machine learning application.

Each of these Application Benchmark's were tested and are presented in a chart containing the absolute benchmark results, the benchmark result normalized against a non-snapshotted & non-introspected baseline, the average memory copy time, and the average dirty page count per snapshot. Each of the next subsections will describe the Application Benchmarks in more detail, describe the performance of the benchmark under snapshotting, and, finally, the performance of the benchmark under simulated introspection.

5.2.1 Kernel Build

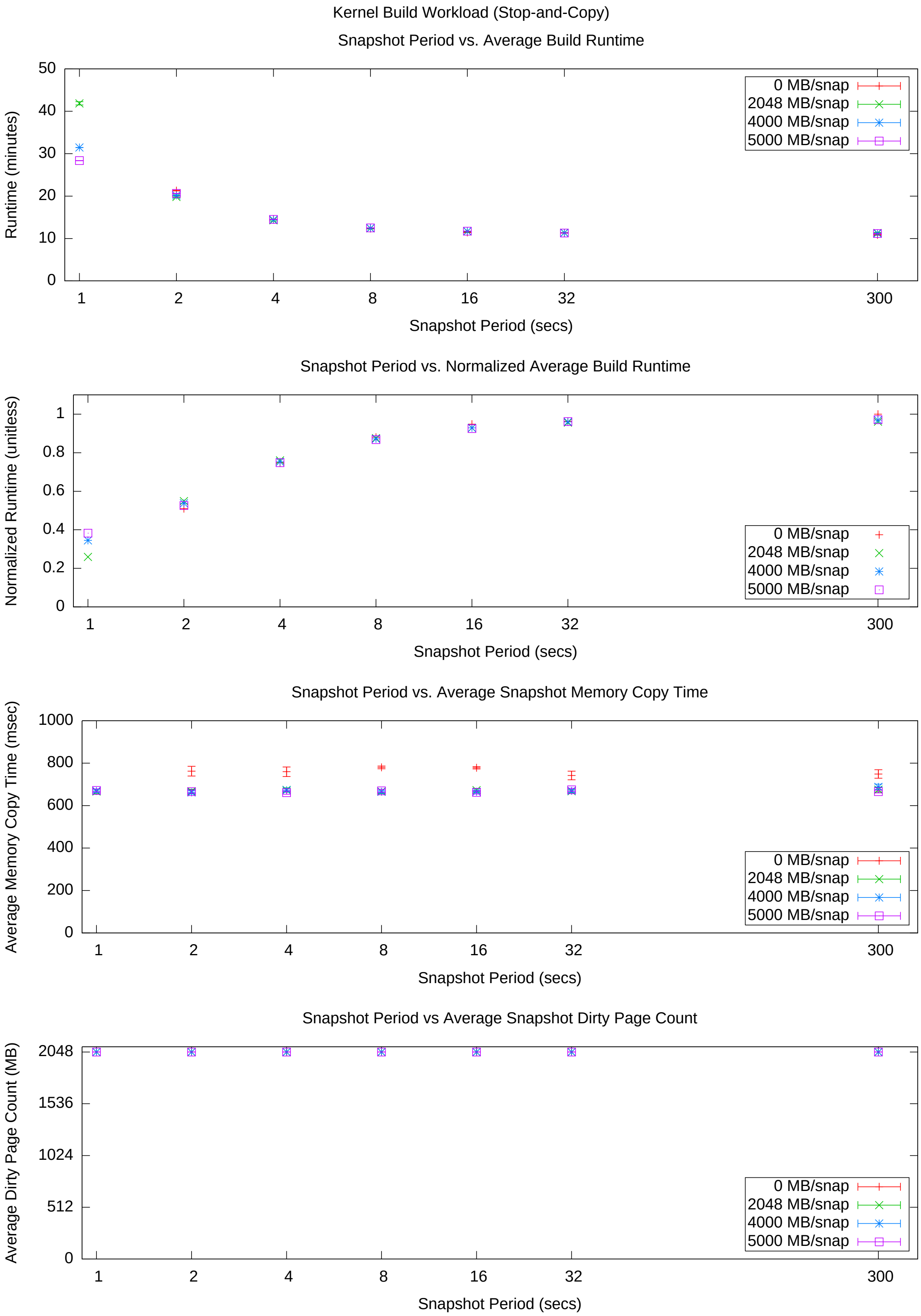
The Kernel Build Application Benchmark consists of building the Linux 3.14 kernel using gcc 4.6.3 in the default configuration (make defconfig). The result of the Kernel Build Application Benchmark is the elapsed build time in seconds. Figure 5.2 illustrates Kernel Build Application Benchmark behavior observed for each of the introspection configurations.

The Kernel Build Application Benchmark completes in approximately 10 minutes absent snapshotting and introspection. The stop-copy snapshot mechanism introduced an normalized overhead of approximately 2-3x at the one second snapshot period but was noisy. The delta-copy snapshot mechanism introduced an overhead of approximately 1.2x. The low average dirty-page count for the snapshots allowed the delta-copy snapshot to be very efficient at reducing snapshot stop times. The effect of the introspection application competeing for memory bandwidth through simulated introspection of the snapshot was minimal, due to the disk- and compute-bound nature of the benchmark.

5.2.2 ClamAV Antivirus Scan

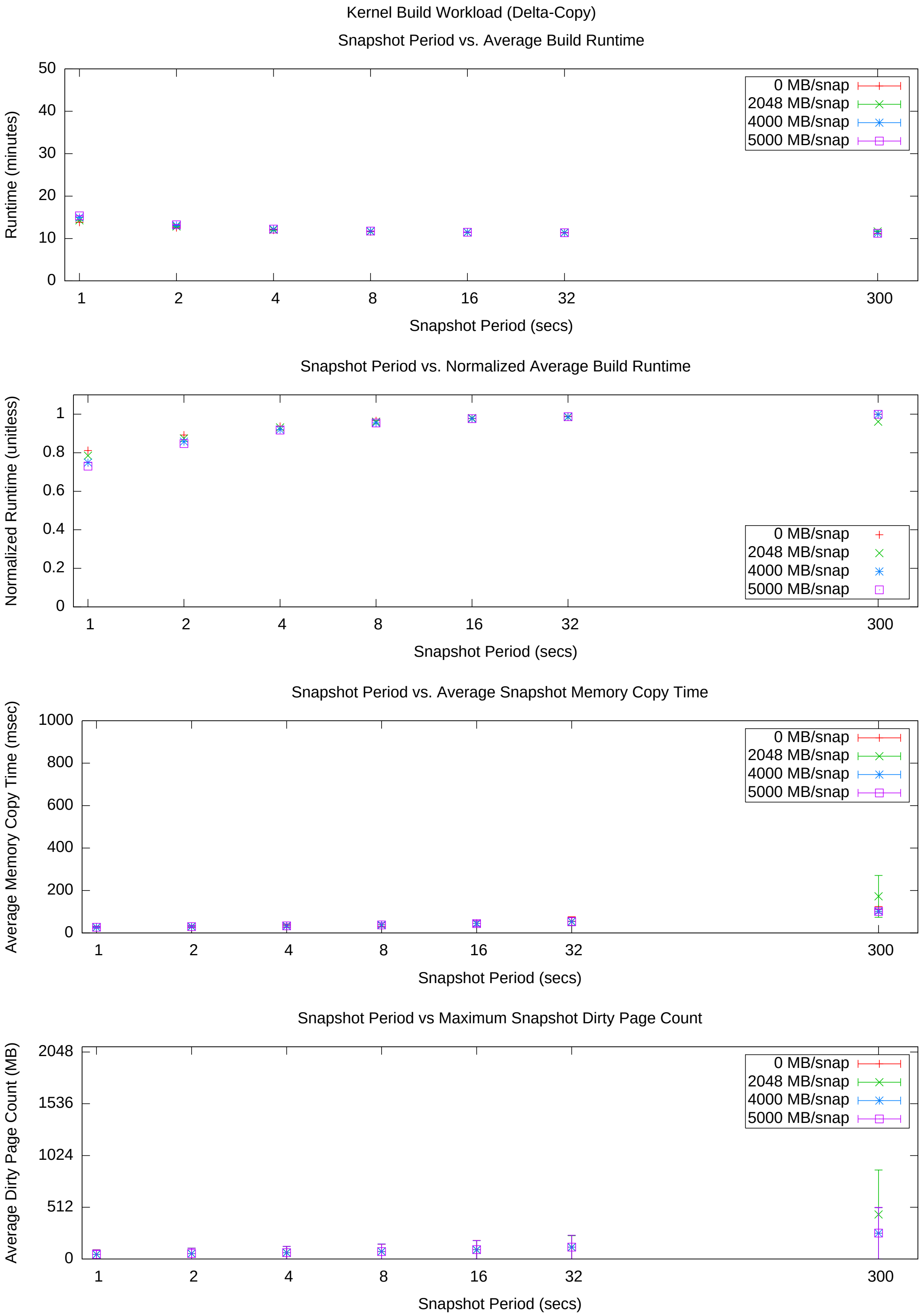
The ClamAV Antivirus Scan Application Benchmark consists the ClamAV Antivirus Scan checking the linux 3.14 source codebase for viruses. The result of the ClamAV Antivirus Scan Application Benchmark is the elapsed scan time in seconds. Figure 5.3 illustrates ClamAV Antivirus Scan Application Benchmark behavior observed for each of the introspection configurations.

The ClamAV Antivirus Scan Application Benchmark completes in approximately 200 seconds absent snapshotting and introspection. The stop-copy snapshot mechanism introduced a normalized overhead of approximately 2.5-4x at the one second snapshot period. The delta-copy snapshot mechanism introduced an overhead of approximately 1.2x. The low average dirty-page count for the snapshots allowed the delta-copy snapshot to be very efficient at reducing snapshot stop times. The ClamAV Antivirus Scan Application Benchmark displays an interesting memory use pattern where the average dirty-page count for short snapshot periods is very low but the dirty-page count for the lifetime of the load is approximately 1.5 GB. The larger snapshot copy times are amortized



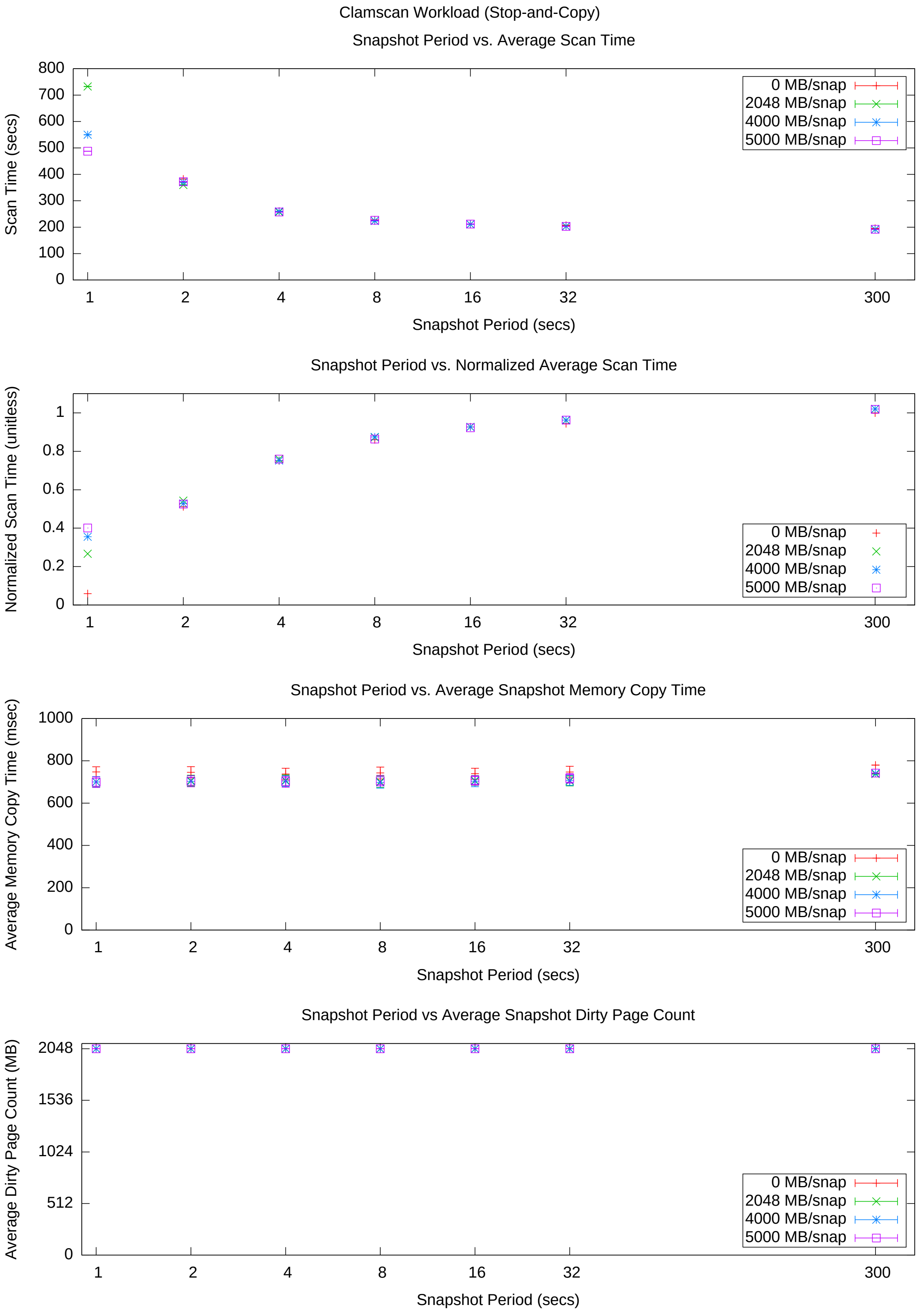
(a) Stop-Copy Snapshot

Figure 5.2: Chart illustrating the Kernel Build Application Benchmark under (a) Stop-Copy and (b) Delta-Copy snapshotting regimes. (Continued on next page.)



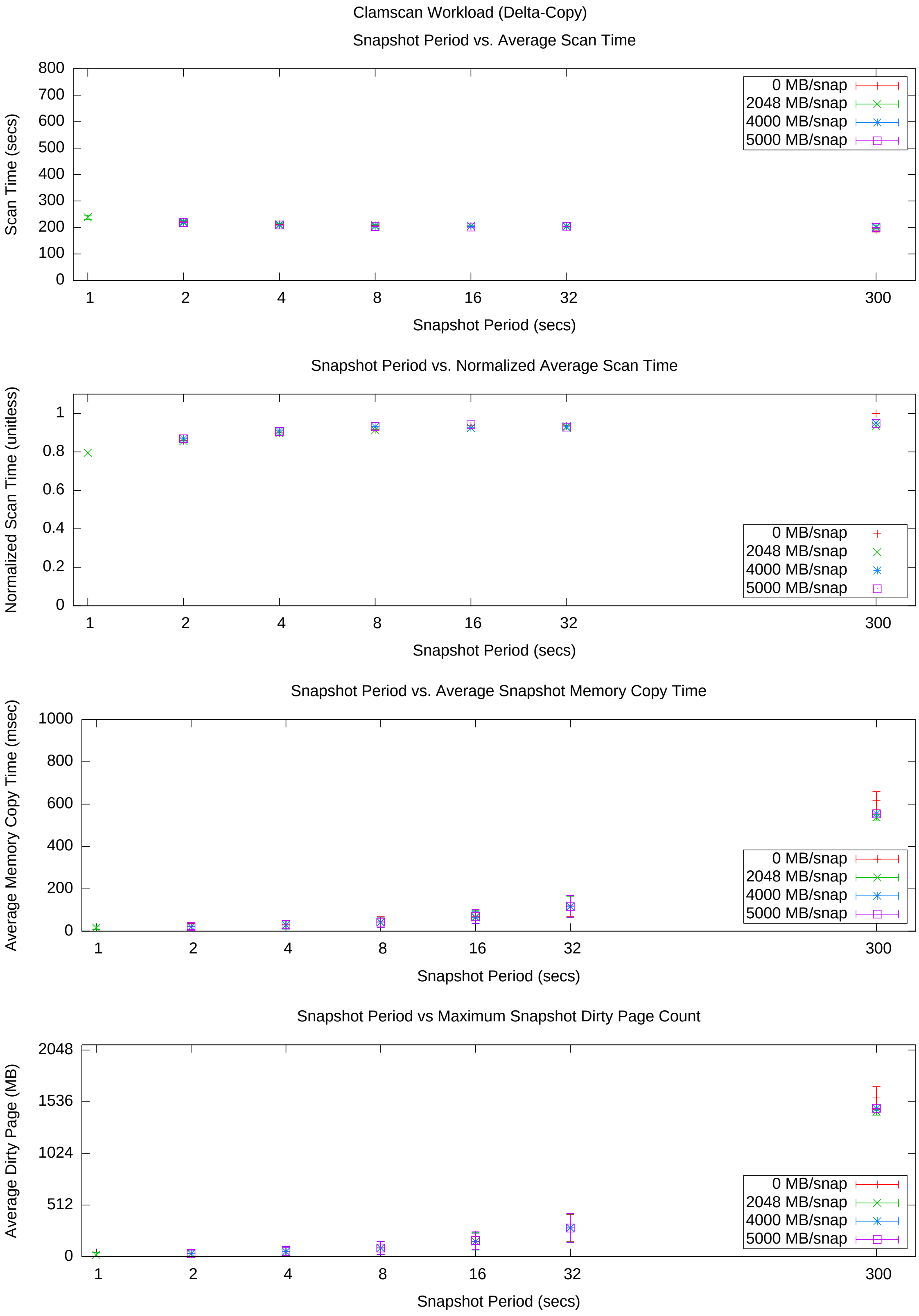
(b) Delta-Copy Snapshot

Figure 5.2: (Continued from previous page.) Chart illustrating the Kernel Build Application Benchmark under (a) Stop-Copy and (b) Delta-Copy snapshotting regimes.



(a) Stop-Copy Snapshot

Figure 5.3: Chart illustrating the ClamAV Scan Application Benchmark under (a) Stop-Copy and (b) Delta-Copy snapshotting regimes. (Continued on next page.)



(b) Delta-Copy Snapshot

Figure 5.3: (Continued from previous page.) Chart illustrating the ClamAV Scan Application Benchmark under (a) Stop-Copy and (b) Delta-Copy snapshotting regimes.

against the longer periods and normalized performance is not affected. The performance of ClamAV Antivirus Scan was only minimally impacted by the simulated introspection.

5.2.3 Apache Web Server

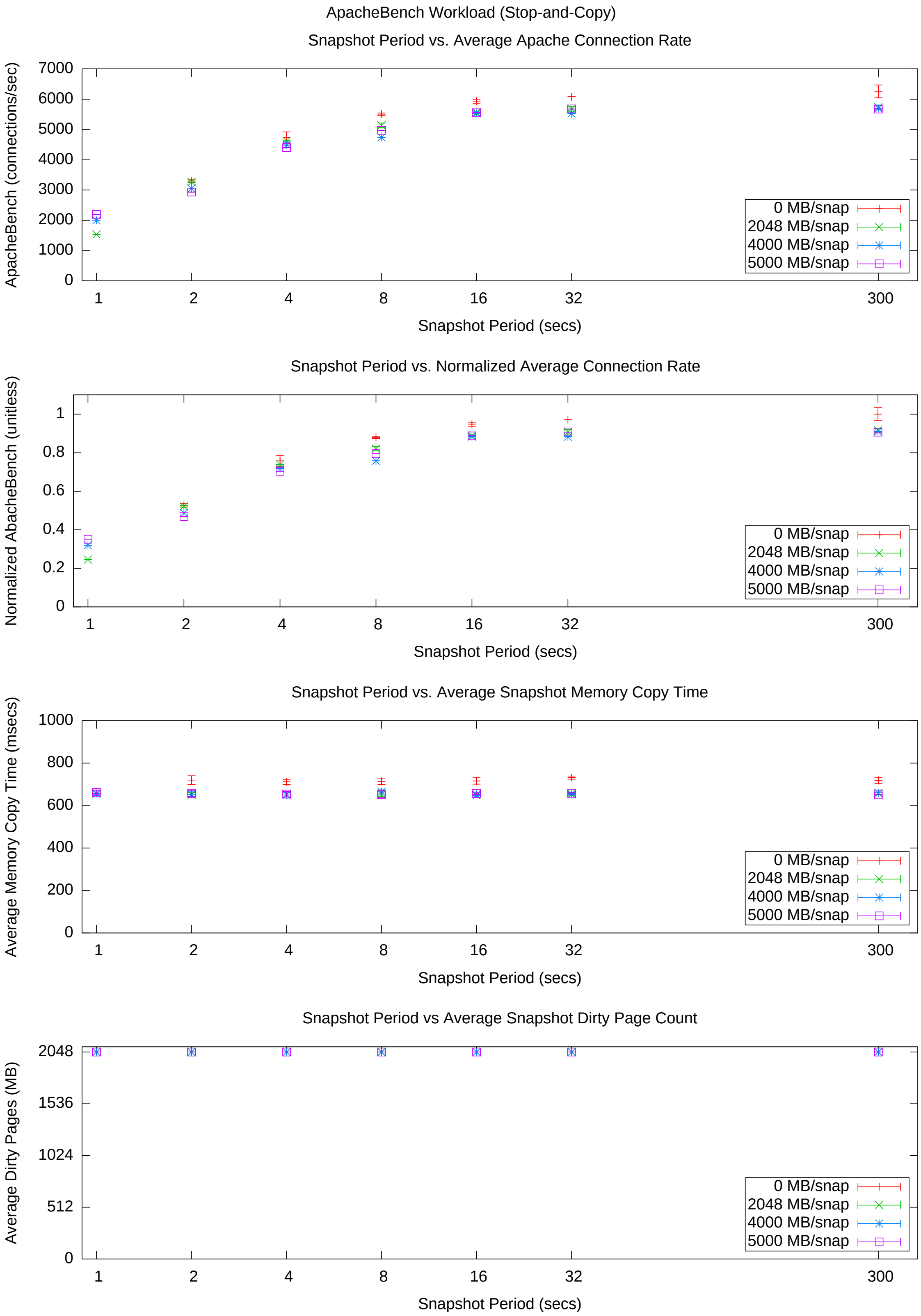
The Apache Web Server Application Benchmark consists of the Apache Web Server running on the introspected guest with a second guest benchmarking it using the Apachebench Apache Benchmark. The result of the Apache Web Server Application Benchmark is the pages served per second by the introspected guest running the Apache Web Server. Figure 5.4 illustrates Apache Web Server Application Benchmark behavior observed for each of the introspection configurations.

The Apache Web Server Application Benchmark is able to handle approximately 5500 connections per second absent snapshotting and introspection. The stop-copy snapshot mechanism causes performance to drop to one quarter of that rate at the one second snapshot period. The delta-copy snapshot mechanism causes performance to drop to eighty percent of that rate at the one second snapshot period. This result agrees well with the observation that the Apache Web Server Application Benchmark dirties less than 64 megabytes of memory under all snapshot periods allowing the delta-copy snapshotting to reduce snapshot stop times. The effect of the simulated introspection of the snapshot on the Apache Web Server Application Benchmark was binary, with a slight jump from no-introspection to any-introspection with no real gradation between the amounts of introspection.

5.2.4 Netperf Network Performance

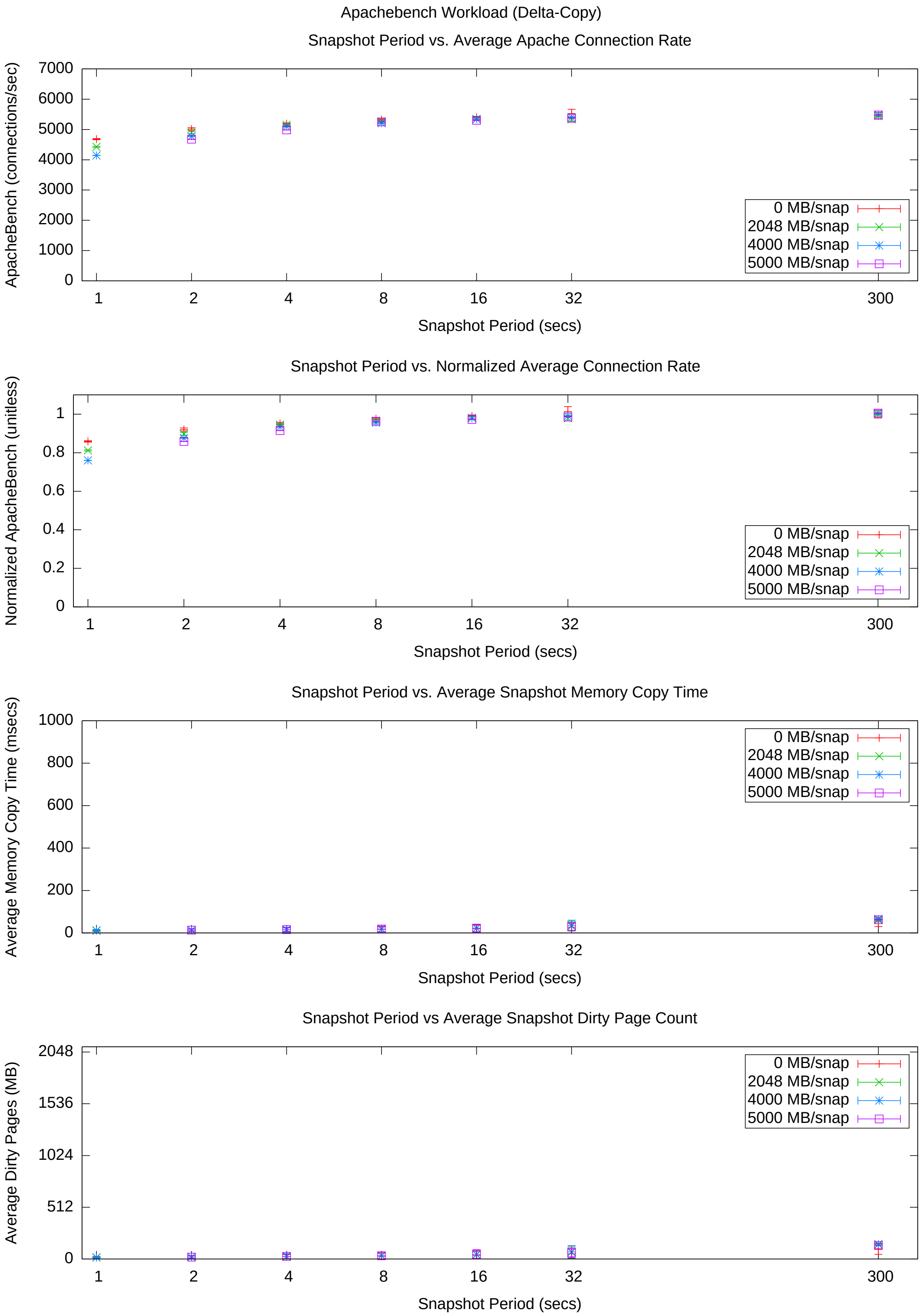
The Netperf Network Performance Application Benchmark consists of the netperf 2.6.0 running on the introspected guest measuring the send packet test speed to a second guest running on the same host. The result of the Netperf Network Performance Application Benchmark is the megabytes per second sent by the introspected guest. Figure 5.5 illustrates Netperf Network Performance Application Benchmark behavior observed for each of the introspection configurations.

The Netperf Network Performance Application Benchmark transfers approximately 6000 megabytes per second absent snapshotting and introspection. The stop-copy snapshot mechanism reduced network transfer performance by fifty percent at the two second snapshot period and the tests failed at the one second period. These failures were due to the snapshotting mechanism not returning from



(a) Stop-Copy Snapshot

Figure 5.4: Chart illustrating the Apache Web Server Application Benchmark under (a) Stop-Copy and (b) Delta-Copy snapshotting regimes. (Continued on next page.)



Apachebench Workload (Delta-Copy)

Snapshot Period vs. Average Apache Connection Rate

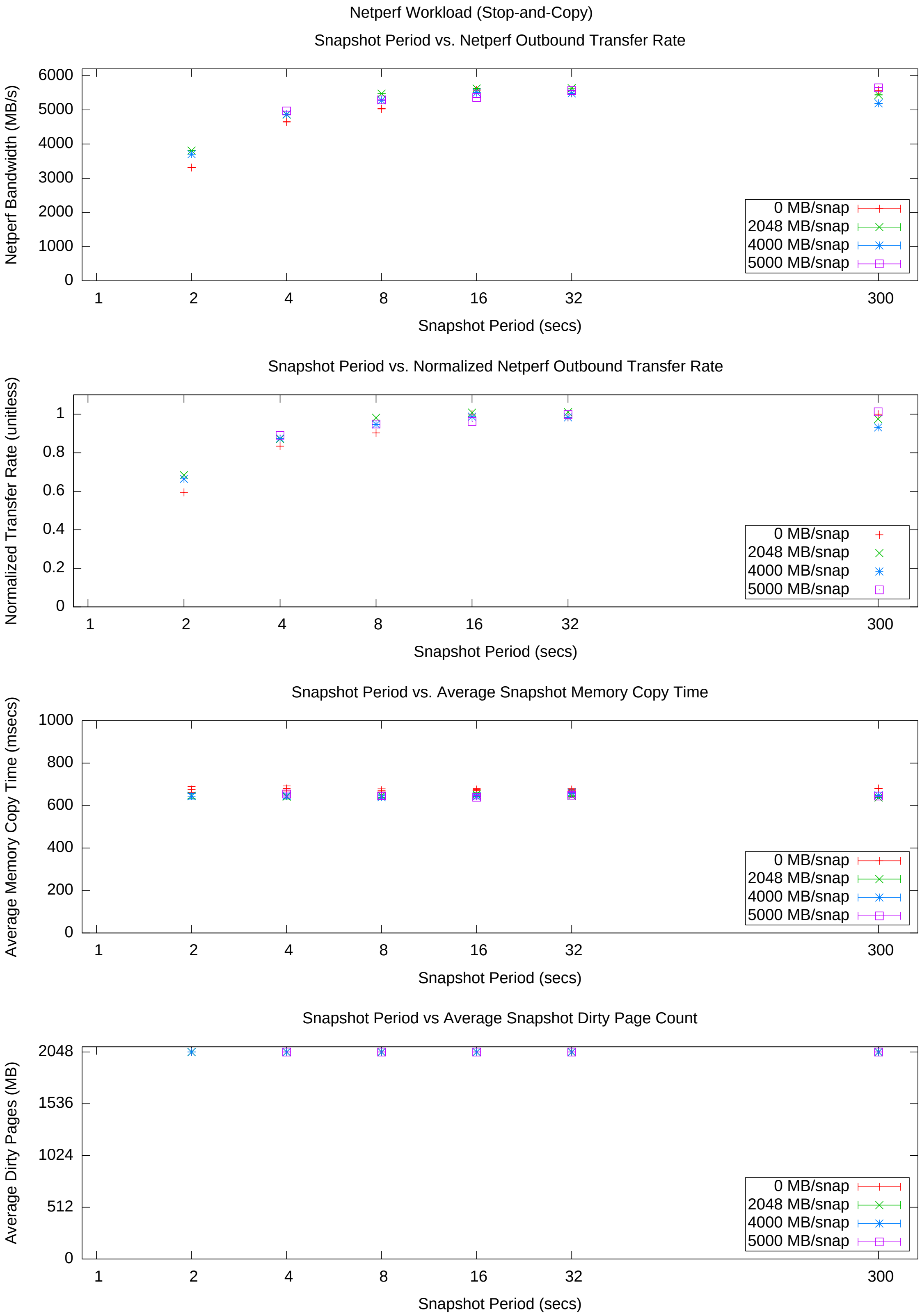
Snapshot Period vs. Normalized Average Connection Rate

Snapshot Period vs. Average Snapshot Memory Copy Time

Snapshot Period vs Average Snapshot Dirty Page Count

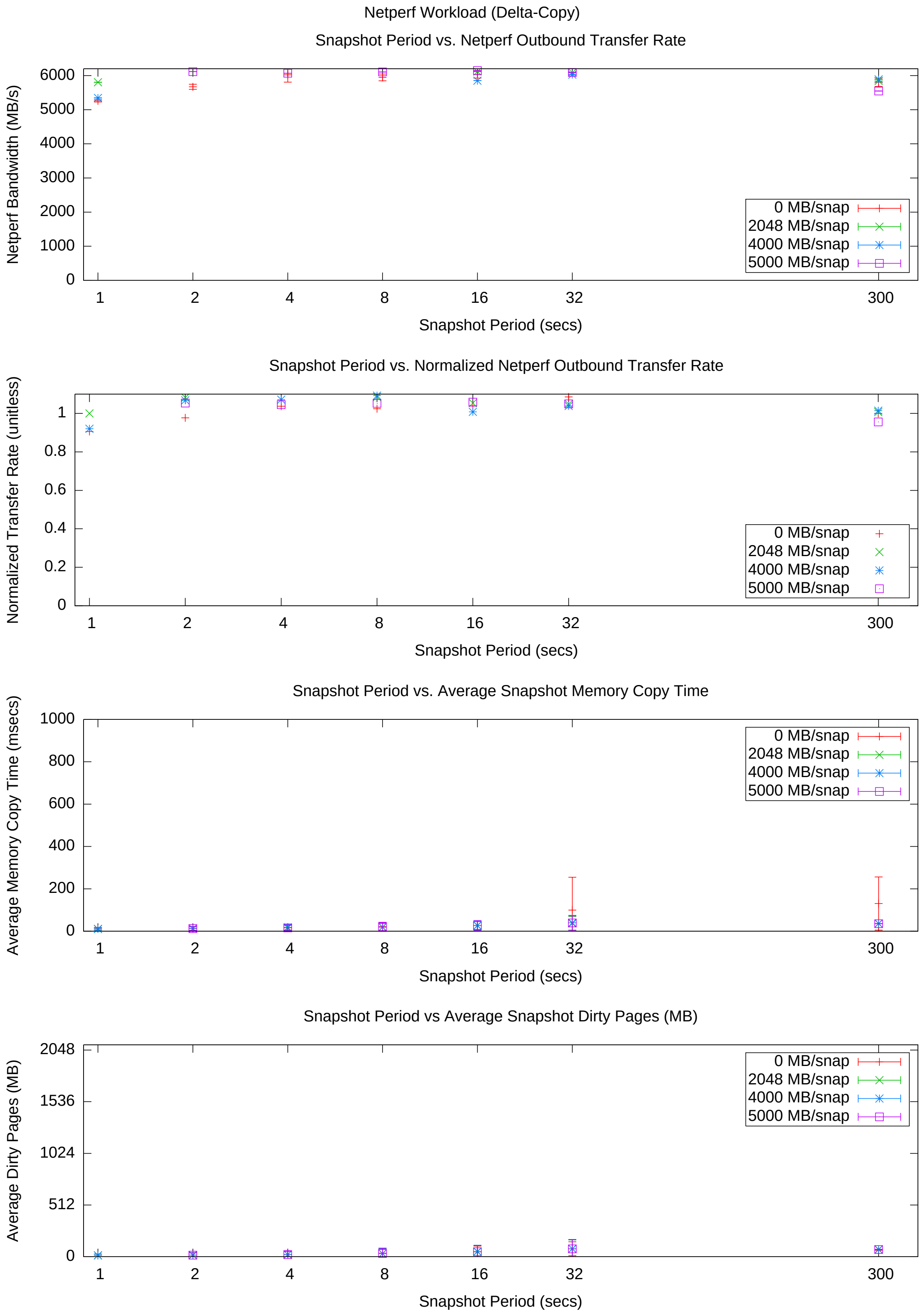
(b) Delta-Copy Snapshot

Figure 5.4: (Continued from previous page.) Chart illustrating the Apache Web Server Application Benchmark under (a) Stop-Copy and (b) Delta-Copy snapshotting regimes.



(a) Stop-Copy Snapshot

Figure 5.5: Chart illustrating the Netperf Network Performance Application Benchmark under (a) Stop-Copy and (b) Delta-Copy snapshotting regimes. (Continued on next page.)



(b) Delta-Copy Snapshot

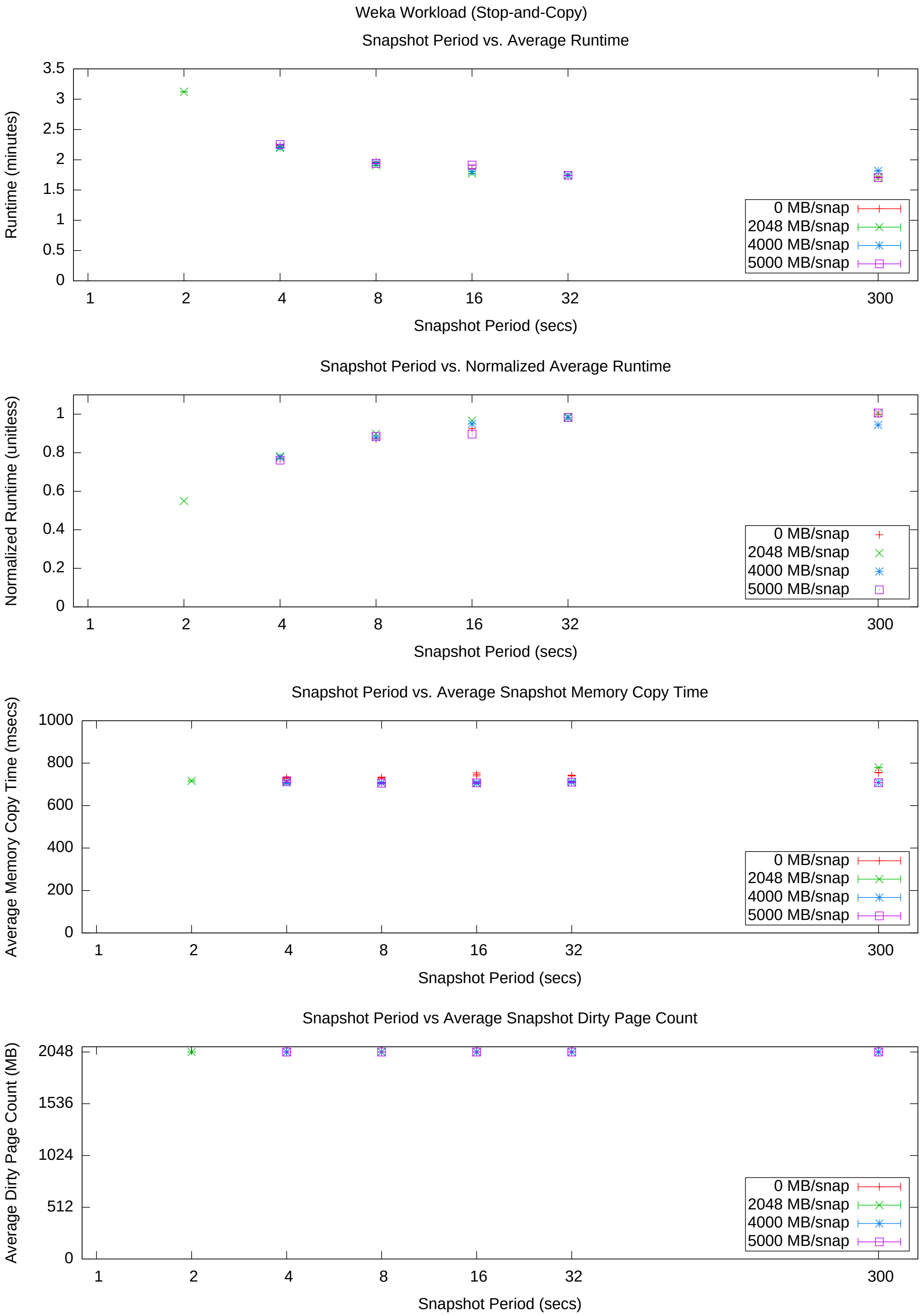
Figure 5.5: (Continued from previous page.) Chart illustrating the Netperf Network Performance Application Benchmark under (a) Stop-Copy and (b) Delta-Copy snapshotting regimes.

the snapshot in time for the next snapshot (i.e. snapshots were taking longer than one second to return). The delta-copy snapshot mechanism reduced performance to approximately 85 percent at the one second snapshot period. The Netperf Network Performance Application Benchmark writes less than 64 megabytes of memory over the benchmarks runtime. The effect of the simulated introspection of the snapshot on the Apache Web Server Application Benchmark was binary, with a slight jump from no-introspection to any-introspection with no real gradation between the amounts of introspection.

5.2.5 Weka Machine Learning

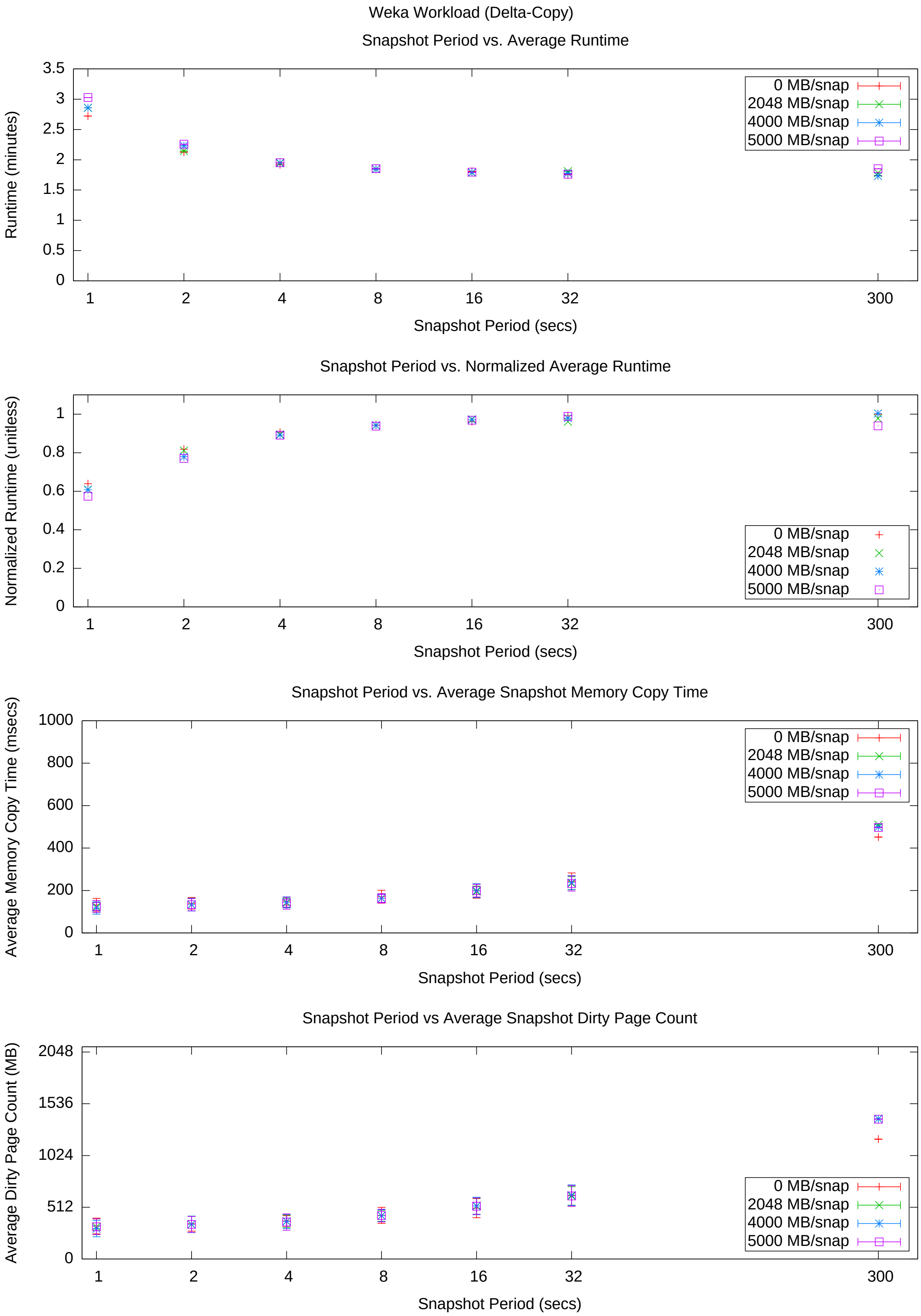
The Weka Machine Learning Application Benchmark consists of the Weka version 3.6.6 SimpleNaiveBayes training and testing on a 300 MB optical character recognition dataset running in the introspected guest. Weka is a Java based tool and the Java VM has been configured with a one gigabyte heap. The result of the Weka Machine Learning Application Benchmark is the time in seconds needed to train the SimpleNaiveBayes model on the training set and then evaluate the test set. Figure 5.7 illustrates Weka Machine Learning Application Benchmark behavior observed for each of the introspection configurations.

The Weka Machine Learning Application Benchmark completes in approximately 100 seconds absent snapshotting and introspection. The stop-copy snapshot mechanism introduced a normalized overhead of approximately 2.5x at the four second snapshot period. The stop-copy snapshot mechanism tests were unable to complete at the one and two second snapshot periods due to failure of the snapshotting mechanism to complete snapshots before the beginning of the next snapshot period. It has been observed that disk-access heavy tests perform very poorly under the current implementation of the prototype and the Weka Machine Learning Application Benchmark contains a period where it loads the 300 MB dataset from the disk into the heap. This behavior may also explain why the delta-copy snapshot mechanism introduced a comparatively large overhead of just over 1.5x at the one second snapshot period. Another explanation for the comparatively slow performance of the Weka Machine Learning benchmark is the relatively large observed average snapshot dirty-page counts that ranged from approximately 256 MB at one second snapshot periods to approximately 1400 MB for 300 second periods. The effect of the simulated introspection application on the Weka Machine Learning was small for all datapoints except for the one second delta-copy period, where



(a) Stop-Copy Snapshot

(b) Chart illustrating the Weka Machine Learning Application Benchmark under (a) Stop-Copy and (a) Delta-Copy snapshotting regimes. (Continued on next page.)



(a) Delta-Copy Snapshot

Figure 5.7: (Continued from previous page.) Chart illustrating the Weka Machine Learning Application Benchmark under (a) Stop-Copy and (a) Delta-Copy snapshotting regimes.

an 15% slowdown is observed when compared to the one second period snapshot-only datapoint.

5.3 Application Benchmarking: Winners & Losers

Delta-copy snapshotting impacted normal guest operation less than stop-and-copy snapshotting. Within delta-copy snapshotting, the applications which performed best under snapshotting and introspection created the fewest dirty pages, like Netperf Network Performance, Apache Web Server, and ClamAV Antivirus Scan, and Kernel Build, when compared to the application that used the most memory, Weka Machine Learning Application Benchmark. Because these applications created fewer dirty-pages, less pages had to be copied at snapshot time, reducing the snapshot stop times and resulting in higher application benchmark performance compared to Weka. These differences are most emphasized at the more frequent snapshot periods then at the longer snapshot periods where the longer stop times are more easily amortized.

Overall the application benchmark evaluation shows that the third requirement for efficient introspection are satisfied for certain applications and introspection configurations. For some applications and introspection configurations, normal guest performance was achieved, but for others performance dropped to a quarter of baseline performance. The next chapter will use microbenchmarking to explore a wider range application properties and introspection configurations to explain Application Benchmark behavior and provide guidance on expected guest load performance.

Chapter 6

Microbenchmark Evaluation

Application Benchmarking provided evidence that normal guest operation could be attained with efficient introspection but only provides insight into the specific guest loads and introspection scenarios tested.

In this chapter, I will expand the range of guest loads and introspection scenarios tested by using two microbenchmarks that mimic critical behavior of the application benchmarks. After systematically evaluating the microbenchmarks under numerous introspection scenarios, some key results are identified that explain how and why efficient snapshotting performs well. Finally, two main factors effecting guest performance are identified and performance guidance is developed for predicting the behavior of guests under introspection scenarios.

6.1 Why Microbenchmarking?

Studying the effect of efficient introspection on the performance of the application benchmark guest loads suggests that the ratio of guest running time to snapshot stop time is the most significant factor influencing guest performance. Snapshot stop time over a period of time is increased by the frequency snapshots are taken and also by the length of time each individual snapshot stops the guest. Microbenchmarking isolates and quantifies the effects various guest load properties have on snapshot stop times.

Specifically, two properties that will be explored are the number of pages that are written by the guest load and the frequency the pages are written (e.g. memory bandwidth utilization). Application benchmarking allowed a limited exploration of these properties, the Weka Machine Learning Application Benchmark exhibited significantly more dirty pages per snapshot than the other application benchmarks. Table 6.1 summarizes the memory write access patterns of the application benchmarks. Microbenchmarking will allow a systematic exploration of the effect of guest load memory access pattern on the performance impact of efficient introspection.

The microbenchmarks complete more quickly, on the order of thirty seconds, than many of the application benchmarks which required many minutes to complete. The faster elapsed execution times allow more microbenchmark evaluations to be performed in limited timeframes.

6.2 Microbenchmark Procedure

Two microbenchmarks were developed to isolate different properties of guest loads. The first microbenchmark, Application Runtime Microbenchmark, measures whether the guest is actively running or not over a period of time. The second microbenchmark is a memory load benchmark that allows various properties of memory access to be exercised on the guest. The block diagram in Figure 6.1 illustrates microbenchmark execution in the guest while the snapshotting and introspection take place on the host.

The same procedure was followed for testing the microbenchmarks as for the application benchmarks. Figure 5.1 illustrates the application benchmark testing procedure. The test begins when the microbenchmark is started in the VM guest, then the guest memory is snapshotted periodically. The application benchmarks are run to completion and the result of the benchmark (runtime, bandwidth,

Application Benchmark	Total Dirty Memory	1 Hz Dirty Memory
Kernel Build	approx. 256 MB	<64 MB
ClamAV Antivirus Scan	approx. 1400 MB	<64 MB
Apache Web Server	<64 MB	<64 MB
Netperf Network Performance	<64 MB	<64 MB
Bonnie++ Disk Performance	<64 MB	<64 MB
Weka Machine Learning	approx. 1400 MB	approx. 256 MB

Table 6.1: Memory access pattern summary for the Kernel Build, ClamAV Antivirus Scan, Apache Web Server, Netperf Network Performance, Bonnie++ Disk Performance, and Weka Machine Learning Application Benchmarks. The approximate dirty page working set size for each application is listed for the complete run of the Application Benchmark and the dirty page working set size for the Application Benchmark when it is sampled at 1 Hz.

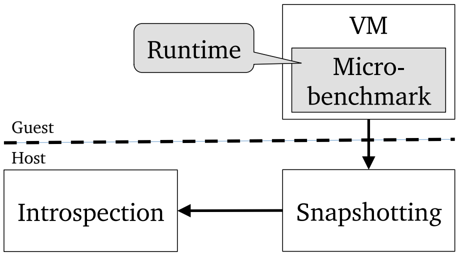


Figure 6.1: Microbenchmarks are used to quickly evaluate the effect of snapshotting over various snapshotting regimes, guest loads, and introspection loads.

Listing 6.1: Application Runtime Microbenchmark validates the snapshot stop times using

```
int main(int argc, char** argv)
{
    parse_args(argc, argv);
    int i;

    int64_t start_time_ms = get_clock_realtime();

    register uint64_t spin_count = 0;
    register uint64_t spin_target = SPIN_COUNT_TARGET;
    for( spin_count = 0; spin_count < spin_target; spin_count++ ) {
    }

    int64_t current_time_ms = get_clock_realtime();

    print_result( current_time_ms - start_time_ms, spin_count );

    return 0;
}
```

requests/second, etc.) is recorded. The test fails if the snapshotting cannot be completed in the period specified between snapshots. The host-guest shared memory is read between snapshots to mimic the behavior of introspection. The test fails if the specified introspection read task cannot complete before the next scheduled snapshot.

6.2.1 Application Runtime Microbenchmark

The Application Runtime Microbenchmark was designed to measure the stop time of the guest independently of the memory bandwidth. To this end, the Application Runtime Microbenchmark attempts to minimize memory utilization by merely incrementing a register to a set limit and then exiting. Pseudo-code for Application Runtime Microbenchmark is in Listing 6.1.

The Application Runtime Microbenchmark microbenchmark can be applied to answering several key questions about the efficient introspection guest. *Is the guest clock trustworthy?* Virtualization systems are notorious for poorly supporting accurate guest time record keeping. By forcing the guest to complete a task of that requires a pre-measured time rather than simply asking the guest to sleep for that time, we can compare the time to complete that task with the host system time and even wall-clock time to verify the guest clock. *What is the overhead of stopping to copy the snapshot?* Because of the complex implementation of the KVM hypervisor, it is not really possible to simply measure the overhead of stopping the guest to copy snapshots. The Application Runtime Microbenchmark measures the time for the guest to complete the spinning task and calculate the overhead imposed by efficient introspection. Answering these questions using the Application Runtime Microbenchmark for each of the introspection scenarios will be a key feature of the microbenchmark evaluation section.

6.2.2 Memory Load Microbenchmark

The Memory Load Microbenchmark measures the effect of varying guest load memory access patterns on efficient introspection. This microbenchmark testing explores a wide range of memory access patterns – including reads and writes, access ranges, and access bandwidth – expanding the understanding of efficient introspection impact on normal guest operation, while isolated from other potential impacts.

Several questions will be answered using the Memory Load Microbenchmark. *How does memory access type affect guest load performance?* Some of the mechanisms involved in snapshotting, specifically delta-copy and pre-copy, may be affected by guest load. Guest loads with more writes will create more dirty pages, dirty pages which must be copied into the snapshot at snapshot stop time. Snapshot stop time is known to impact guest performance. *How does memory bandwidth load affect application runtime?* Copying snapshot memory requires access to the limited memory bandwidth of the virtualization platform and may compete with the guest resources. The Memory Load Microbenchmark will help us answer these questions among others.

The Memory Load Microbenchmark is a modified version of the `lmbench bw_mem` microbenchmark version 3.0-a9 by Staelin [13]. In unmodified state, the `bw_mem` parameterizes the working set size of the bandwidth test and measures the maximum bandwidth of reads or writes that can be

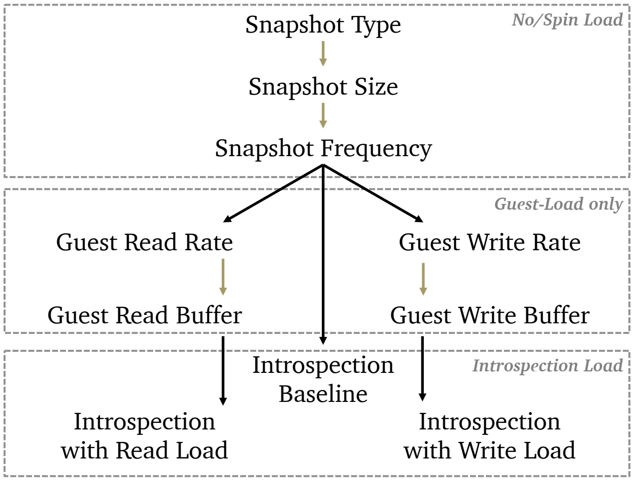


Figure 6.2: The Microbenchmarks are evaluated against the varying snapshot and introspection regimes according to the above strategy. First, the snapshot-related parameters are tested against the Application Runtime Microbenchmark in the "No/Spin Load" tests. Next, the "Guest-Load Only" tests evaluate the effect of snapshotting on the Memory Load Microbenchmark for various configurations. Finally, the "Introspection Load" tests measure the effect of simulated introspection on the performance of the Memory Load Microbenchmark.

written into a buffer of that size. The Memory Load Microbenchmark was developed by creating a further parameter, memory access bandwidth, that allows various bandwidths to be generated by inserting point operations between the memory accesses. These floating point operations have the effect of slowing the rate at which memory can be accessed. Various bandwidth setting were created that slowed the memory access to roughly various amounts. These bandwidth settings could then be used to mimic the behavior of various guest applications.

6.3 Microbenchmark Evaluation

Microbenchmark evaluation provides an opportunity to demonstrate the performance impact of efficient introspection on normal guest operation over a variety of guest loads and introspection scenarios. Figure 6.2 summarizes the strategy I will employ for systematically exploring the parameter space of the snapshotting regimes, guest loads, and introspection in order to isolate the performance effects. To this end, evaluation is broken into three phases: first, "No/Spin" load where the snapshotting parameters are evaluated without any guest load or introspection; second, the "Guest-Load Only" phase, where the Memory Load Microbenchmark-loaded guest is evaluated under various snapshotting configurations; and, finally, the "Introspection Load" phase, where the Memory Load

Microbenchmark-loaded guest is evaluated under snapshotting and simulated introspection loads. The snapshot regimes include the snapshot type (stop-copy, delta-copy, and pre-copy, the snapshot size (sometimes varied, but usually set at 2 GB), and snapshot period (as low as 1/4 s). The guest loads explored will include read and loads with varying buffer sizes and access rates. The introspection loads mimic the effect of introspection application by reading from the guest snapshot at various rates between 1 and 8 GB/s. The rest of this section will explore guest load performance under these configurations.

6.3.1 Stop-Copy Snapshot Evaluation

The stop-and-copy snapshot mechanism is mechanically the simplest and a good place to begin evaluation. Further, stop-copy will be used as a basis of comparison with other snapshotting regimes in later evaluation. The performance of stop-copy is evaluated in the absence of a guest load, then various guest loads will be evaluated with snapshotting, and finally the effect of simulated introspection is introduced.

Stop-Copy: No Load/Spin Load

Snapshotting consists of pausing the guest, copying out the snapshot memory, and then restarting the guest. First I will measure the time to copy memory for a stop-copy snapshot and then examine the full impact of snapshotting on guest performance. Figure 6.3 illustrates the snapshot memory copy time for an unloaded and Application Runtime Microbenchmark-loaded guest. A variety of snapshot sizes between 0 and 2048 MB are shown and the snapshot memory copy times range from nearly zero milliseconds for the zero MB snapshot to approximately 700 milliseconds for the 2048 MB snapshot. Both the ”No Load” and Application Runtime Microbenchmark ”Spin Load” can be observed to follow nearly identical behavior, suggesting that the Application Runtime Microbenchmark load does not affect the stop-and-copy snapshot memory copying mechanism. The memory copy rate observed in this test is approximately 3000 MB/s which is substantially lower than the best rate for memory copy observed on this guest (approximately 7000 MB/s). This is due the limitations of using the built-in KVM memory copying capabilities that ensure a coherent memory snapshot.

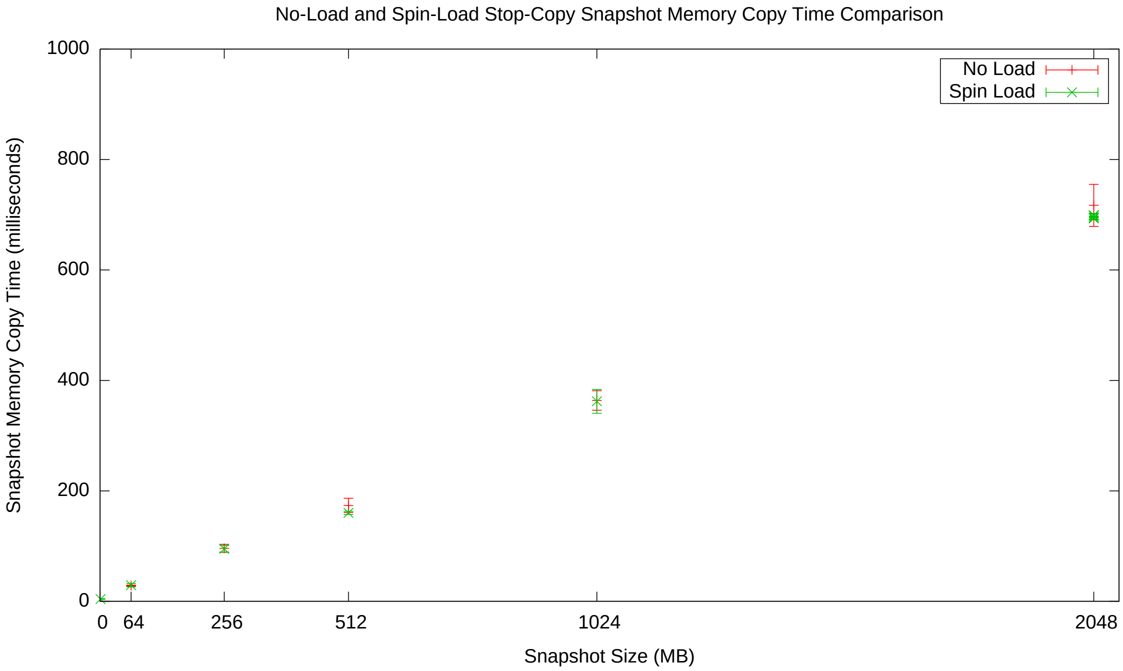


Figure 6.3: Stop-copy snapshot memory copy time for various size snapshots of an unloaded guest and of an Application Runtime Microbenchmark-loaded guest.

Now that it has been established that stop-and-copy snapshot copy time is not affected by the Application Runtime Microbenchmark, the total overhead of the snapshots can be measured. Figure 6.4 illustrates the run time overhead of stop-copy snapshots on a Application Runtime Microbenchmark-loaded guest that is being stop-copy snapshotted at one Herz. The accounting is broken down into three parts: the base spin runtime, the baseline runtime of the Application Runtime Microbenchmark load; the memory copy time, the snapshot memory copy time directly measured by the guest; and, finally, the unaccounted stop time, which is the time leftover from the total guest load runtime less the base time and the memory copy time.

In addition to snapshot size, the effect snapshot frequency on the stop-and-copy mechanism was investigated. Figure 6.5 illustrates the effect of snapshot frequency on the snapshot memory copy time on an unloaded guest. Two snapshot sizes were investigated (600 MB and 2048 MB) and no change in snapshot memory copy times was observed across a range of snapshot periods (0.5 s to 32.0 s).

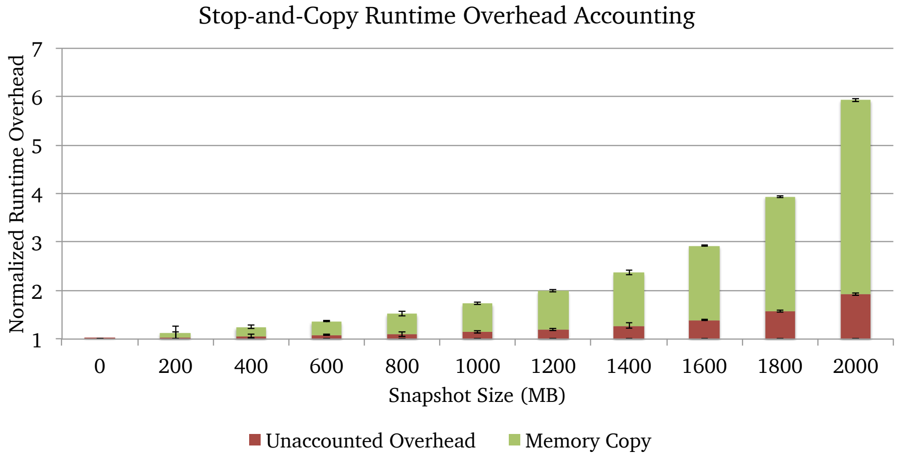


Figure 6.4: Accounting for Stop-Copy run time overhead in varying sized guests.

Stop-Copy: Guest Load

After evaluating the stop-copy snapshot mechanism on an essentially unloaded guest, the stop-copy snapshotting is now studied in the context of a Memory Load Microbenchmark-loaded guest. Figure 6.6 illustrates runtime overhead of stop-copy snapshotting in a wide variety of circumstances. Figure 6.6(a) contains six charts, each presenting the normalized runtime of the Memory Load Microbenchmark reading from three working-set-size configurations (64, 512, and 512 MB). The first five charts present the normalized access performance of the benchmark while being snapshotted at varying periods (1, 2, 4, 8, and 16 seconds) compared to the access speed of that benchmark configuration in baseline (no-snapshotted) configuration. The 128.0 second snapshot period chart differs differs from the others because it presents the absolute performance of the benchmark against the baseline performance.

The baseline chart is changed in this way to illustrate that at 128.0 second snapshot period, the benchmark performs at baseline level. As the frequency of snapshotting increases, or the period decreases, the performance of the efficient introspection can be observed to decrease. The decrease is flat across the configured benchmark access speeds, suggesting that stop-copy snapshot stop time is the cause of the slowdown rather than memory bandwidth bottlenecks. This is borne out by the intuition that snapshot stop times are relatively long (tenths of seconds) events and that the snapshot only copies memory while the guest is halted, meaning that guest and snapshot memory requests are seperated temporally and that they are not in competition. Finally, the working-set-size and access

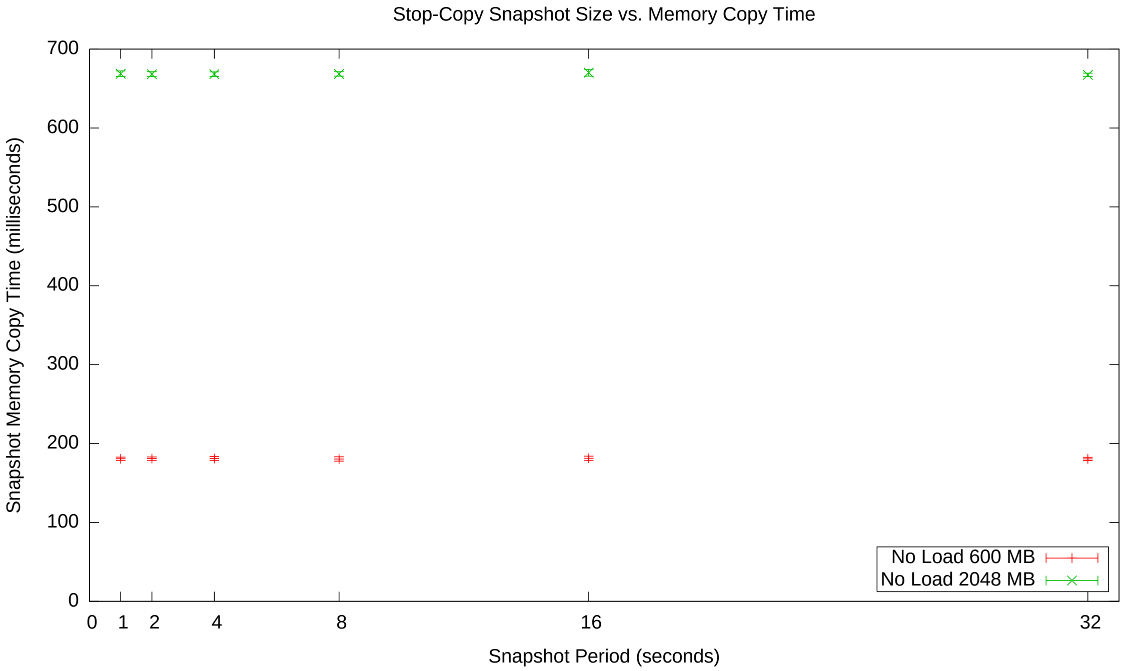


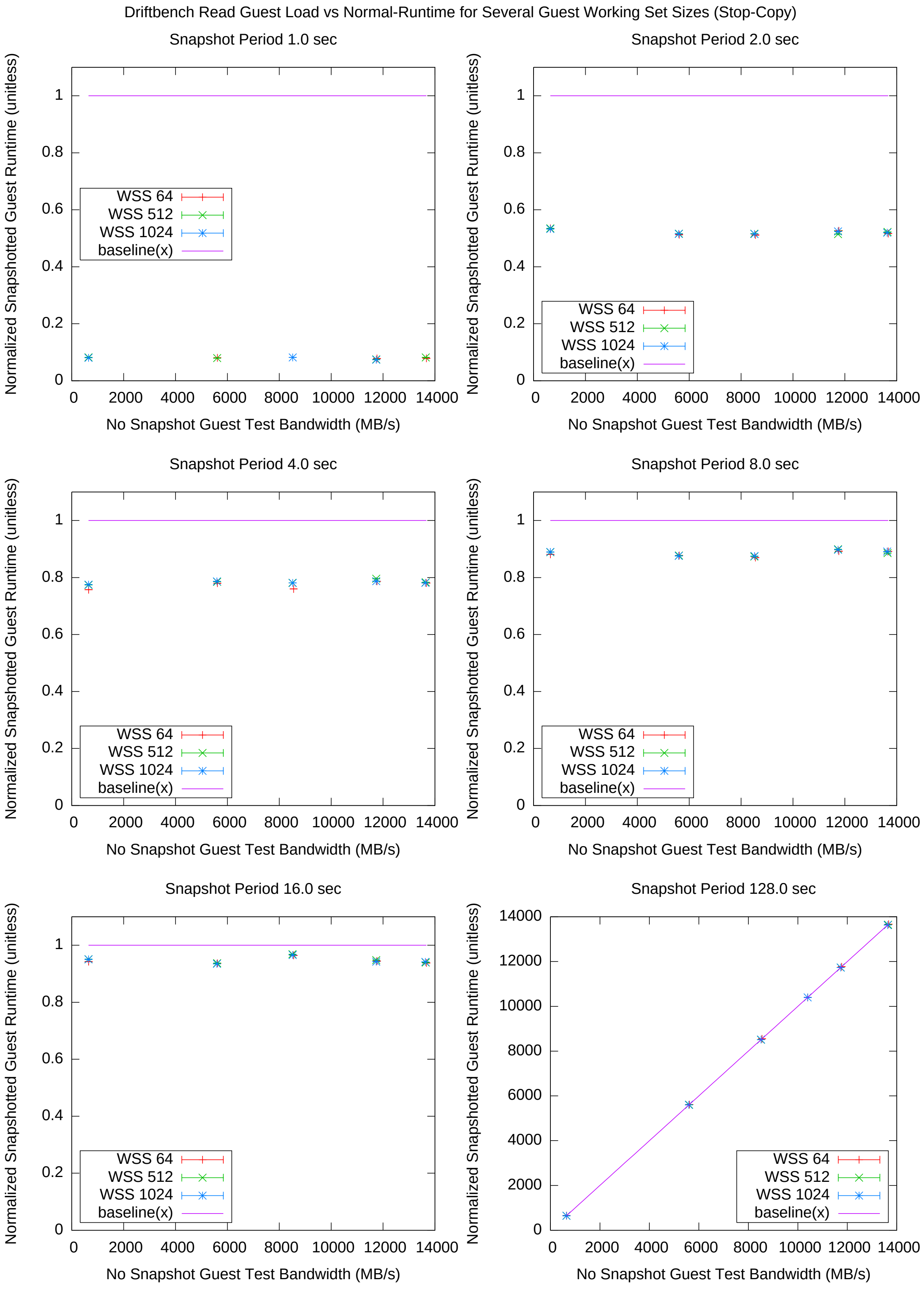
Figure 6.5: Effect of snapshot period on snapshot memory copy time for variously-sized unloaded guests.

rate had no observable effect on the read-performance of the benchmark. Stop-copy snapshotting copies all memory regardless of whether it had been read previously.

The performance of the read benchmarks is very similiar to the performance of the write benchmarks. Figure 6.6(b) contains a similiar six charts, but with the write-load instead. Again, baseline write performance is observed at the 128.0 second snapshot period. Snapshot period is related to performance with one second snapshot period correlating to performance drops of over ninety percent. The performance impact of stop-copy snapshotting is flat across memory access speeds for specific snapshot periods, suggesting that only stop-time is impacting benchmark performance. Finally, working-set-size and access rate had no observable effect on write benchmark performance, only snapshot frequency.

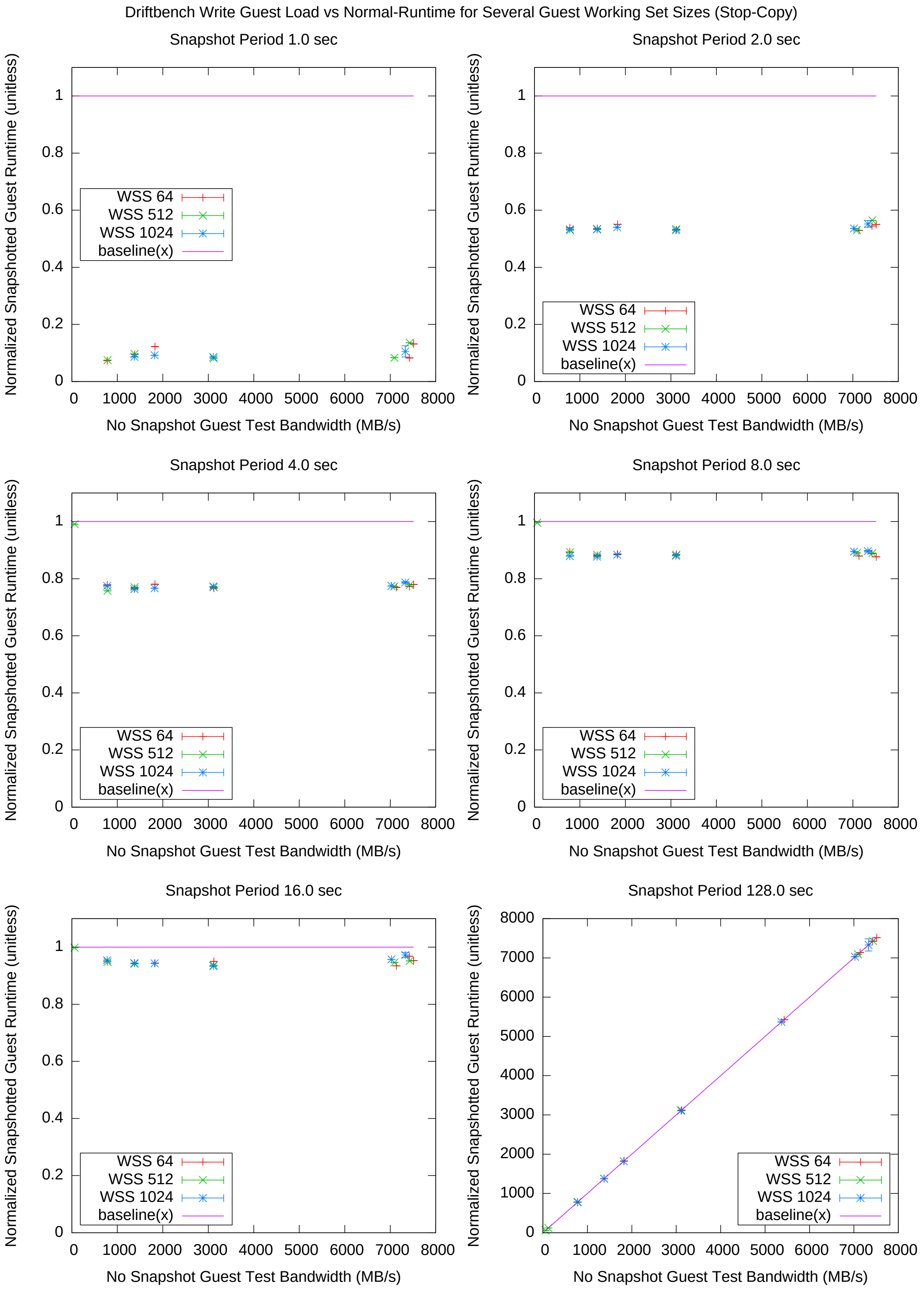
Stop-Copy: Introspection Load

After examining the effect of the guest-load on snapshotting, we now add simulated-introspection loads to the snapshotted-guest load scenario. Figure 6.7 illustrates the runtime overhead of stop-copy snapshotting and introspection on a Memory Load Microbenchmark-loaded guest. The figure



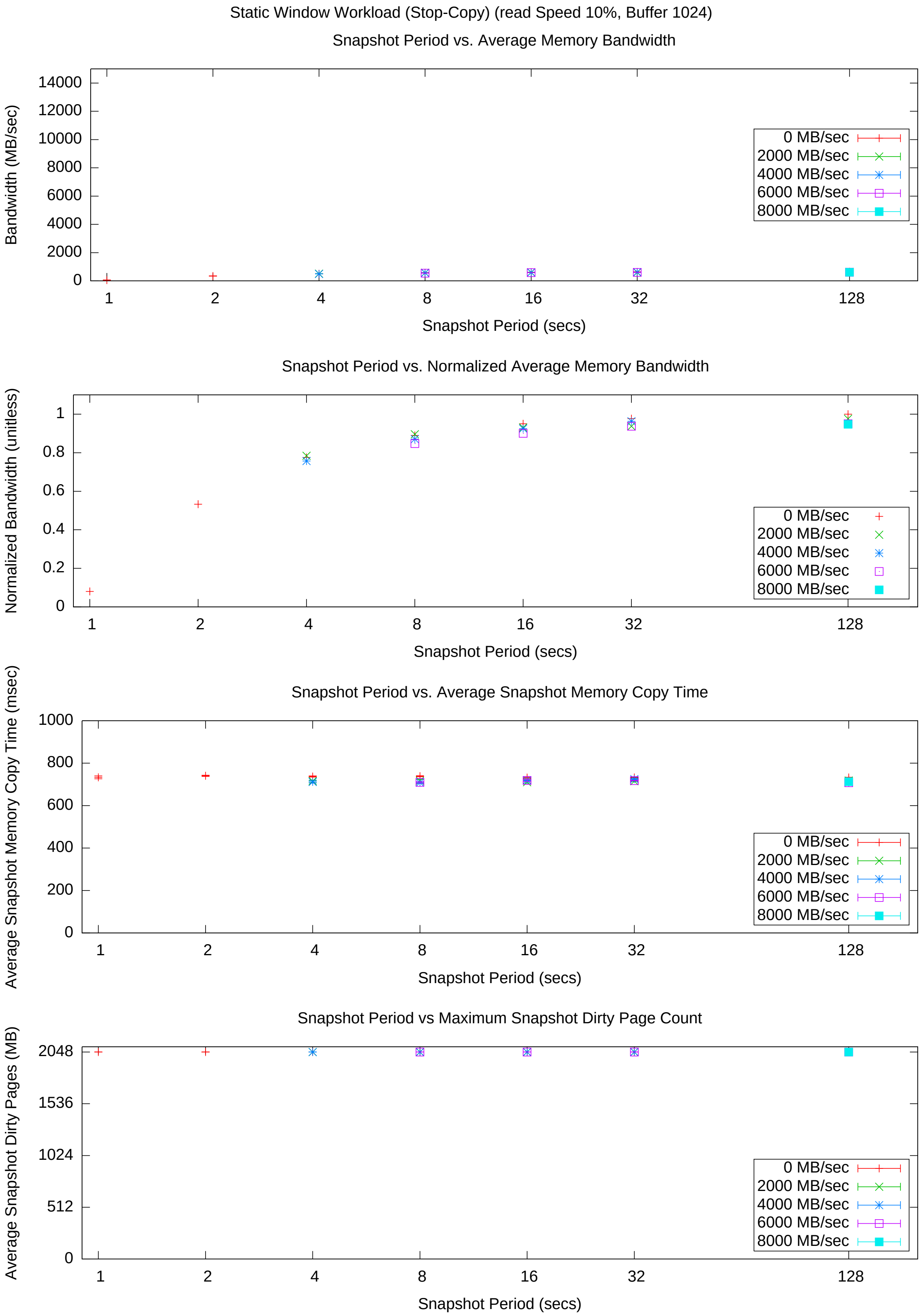
(a) Read Load

Figure 6.6: Runtime overhead of Stop-Copy Snapshotting on (a) read and (b) write guest loads with varying working set sizes and access rates. (Figure continues on next page.)



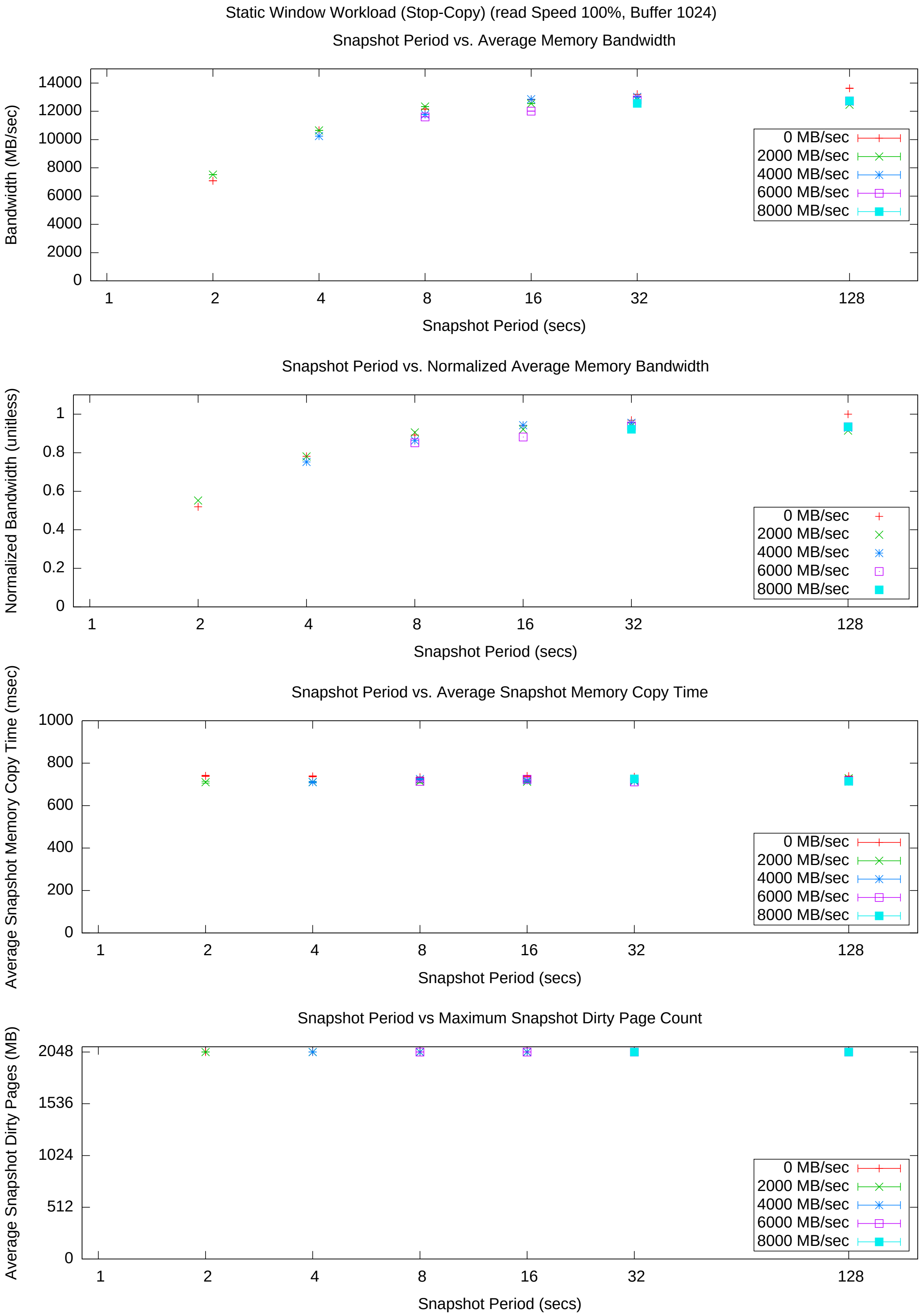
(b) Write Load

Figure 6.6: Runtime overhead of Stop-Copy Snapshotting on (a) read and (b) write guest loads with varying working set sizes and access rates. (Figure continued from previous page.)



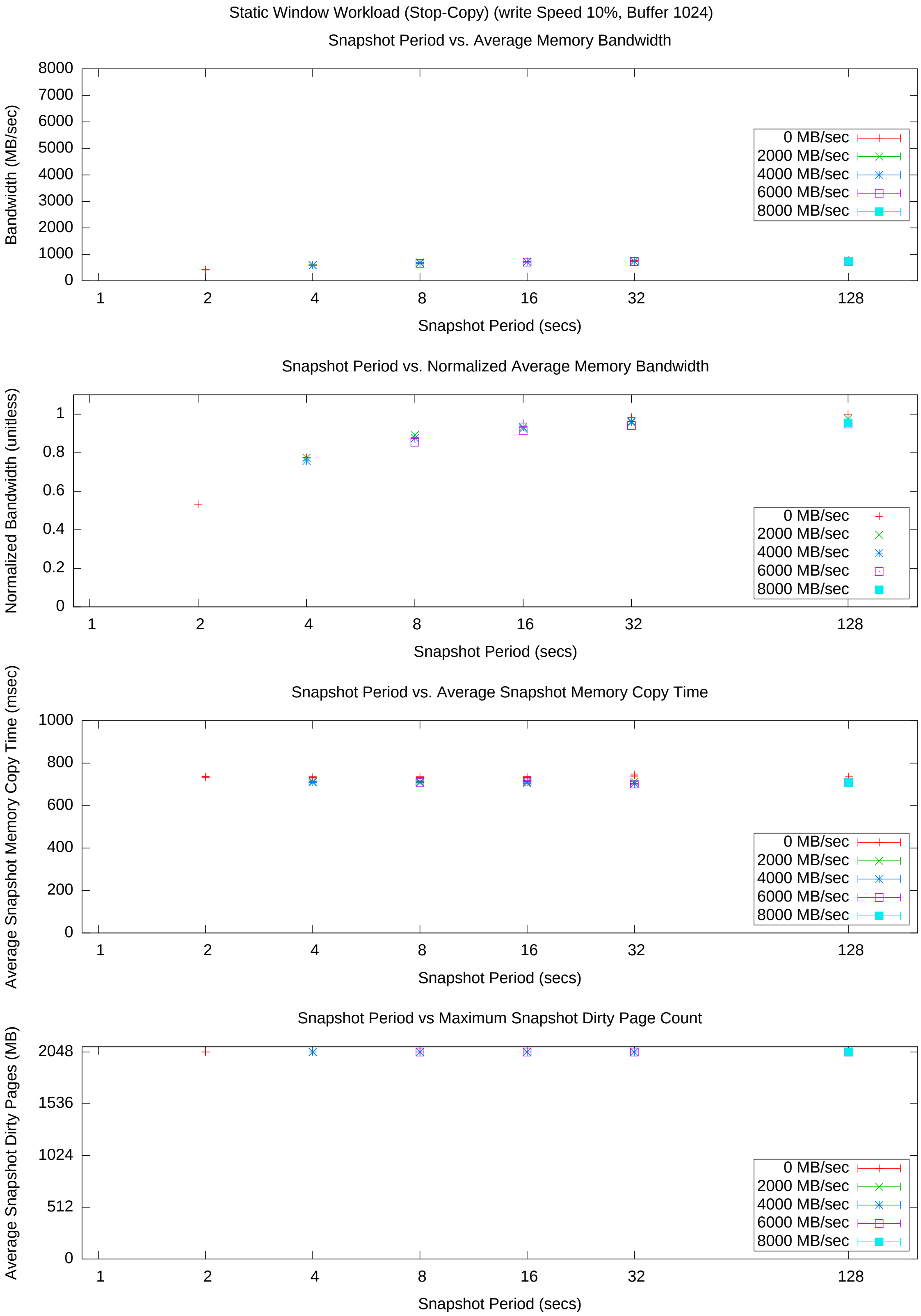
(a) Slow-Read Load

Figure 6.7: Runtime overhead of Stop-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with snapshot period. Fast accesses at maximum rate possible and slow accesses rate limited to ten percent of maximum. (Figure continued on next page.)



(b) Fast-Read Load

Figure 6.7: Runtime overhead of Stop-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with snapshot period. Fast accesses at maximum rate possible and slow accesses rate limited to ten percent of maximum. (Figure continued on next page.)



Static Window Workload (Stop-Copy) (write Speed 10%, Buffer 1024)

Snapshot Period vs. Average Memory Bandwidth

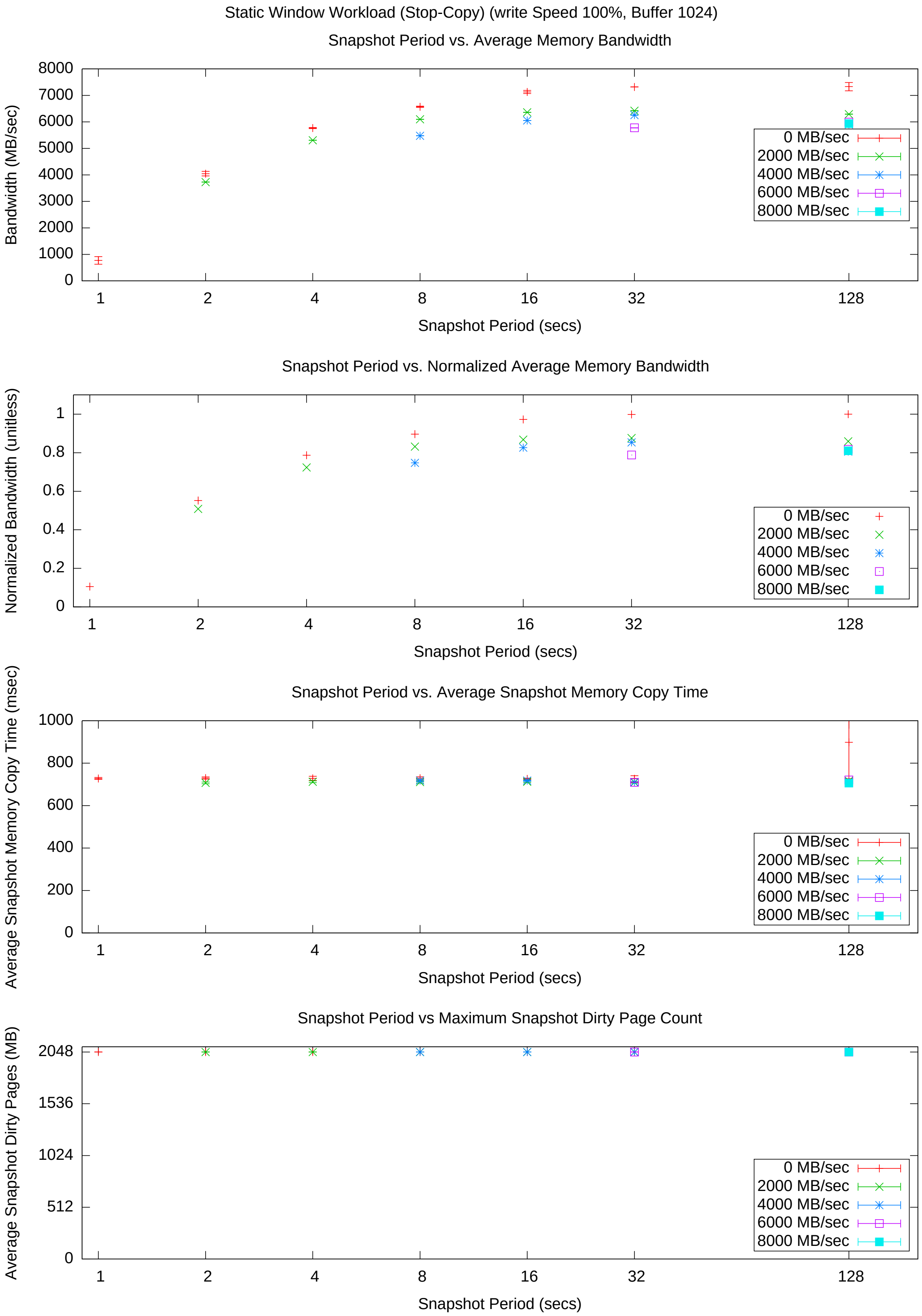
Snapshot Period vs. Normalized Average Memory Bandwidth

Snapshot Period vs. Average Snapshot Memory Copy Time

Snapshot Period vs Maximum Snapshot Dirty Page Count

(c) Slow-Write Load

Figure 6.7: Runtime overhead of Stop-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with snapshot period. Fast accesses at maximum rate possible and slow accesses rate limited to ten percent of maximum. (Figure continued on next page.)



(d) Fast-Write Load

Figure 6.7: Runtime overhead of Stop-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with varying access rates. The fast accesses were performed at the maximum rate possible and the slow accesses were rate limited to ten percent of the maximum. (Figure continued from previous page.)

contains four subfigures that represent the four test series that were undertaken, Figure 6.7(a) shows a 10% baseline performance slow-read load, Figure 6.7(b) shows a 100% baseline performance fast-read load, Figure 6.7(c) shows a 10% baseline performance slow-write load, and Figure 6.7(d) shows a 100% baseline performance fast-write load. Each of these subfigures contains four graphs: the top graph shows the absolute benchmark performance versus snapshot period, the second graph shows the normalized benchmark performance versus snapshot period, the third graph shows average snapshot memory copy time versus snapshot period, and the bottom graph shows the average dirty pages copied each snapshot versus snapshot period.

The test series measure the impact of introspection by reading from the snapshot at varying rates (baseline 0 MB/s, 2000 MB/s, 4000 MB/s, 6000 MB/s, and 8000 MB/s). The introspection rates are not dynamically tailored like with the Memory Load Microbenchmark. Instead the introspection mechanism is tasked with reading a certain number of megabytes per snapshot period. For example, in the case of the two second snapshot period and 6000 MB/s introspection load, the introspection mechanism would attempt to read 12000 MB (6000 MB/s for 2 seconds) between each snapshot. If the introspection mechanism cannot complete this assigned read task, the test is abandoned and no result is recorded. This effect can be observed in all four test-series, as the test periods became shorter, the ability to perform introspection reduced. At shorter snapshot periods, the time spent actually snapshotting, wherein the introspection cannot take place, becomes a significant barrier to completing the simulated-introspection task.

Several interesting results can be observed from the four introspection test series that were undertaken. First, introspection has little impact on the read efficient introspection. The performance of the microbenchmark is very similiar no matter whether the read access rate was 10% or 100%. Second, guest fast write performance is negatively impacted by introspection competition. The slow (10%) write rates were not impacted by the introspection at any rate, but the fast (100%) write microbenchmark results with any level of introspection can be seen to slow to 80% of the baseline. This slowdown is observed to different degrees across all snapshot periods for the fast write microbenchmark. This result suggests that the guest-load is competeing for memory bandwidth with the introspection load that is running simulataneously.

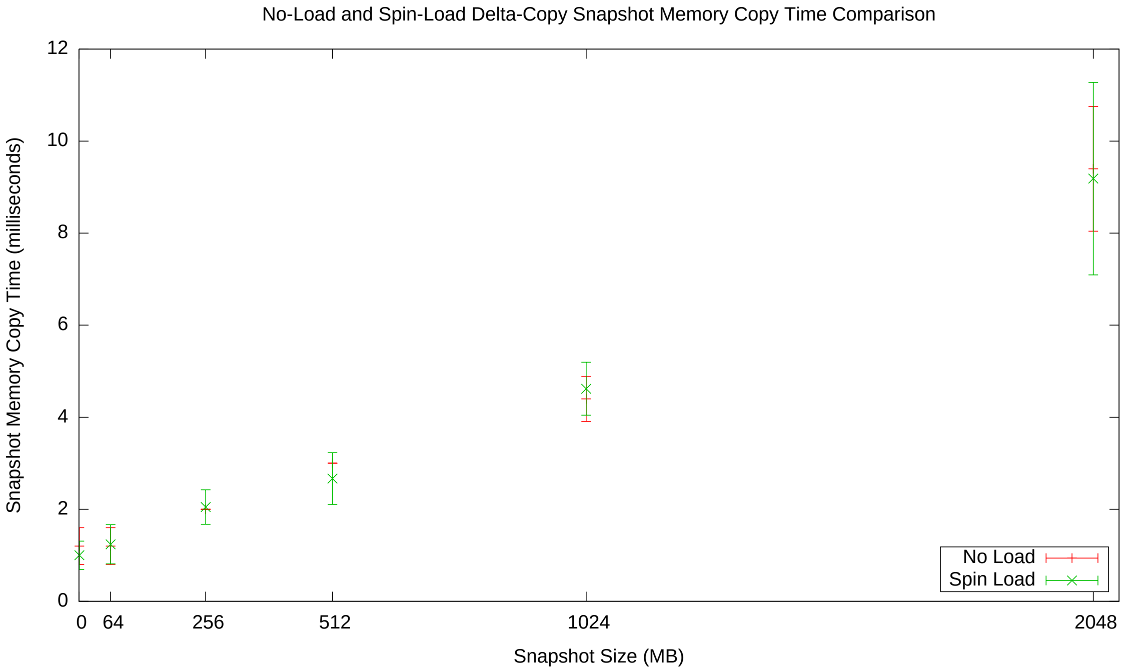


Figure 6.8: Delta-Copy snapshot size versus snapshot memory copy time for an unloaded guest and an Application Runtime Microbenchmark (spin)-loaded guest.

6.3.2 Delta-Copy Snapshot Evaluation

The Delta-Copy snapshot mechanism offers increased efficiency over the Stop-Copy snapshot mechanism by only copying memory pages that have been changed by the guest since the previous snapshot. This increased efficiency brings a dependency between the guest load behavior and the snapshotting overhead. Evaluation will begin with the Delta-Copy mechanism in the absence of a guest load, then various guest loads will be evaluated with snapshotting, and finally the effect of introspection will be introduced.

Delta-Copy: No Load/Spin Load

The delta-copy snapshot mechanism only copies pages that have changed since the previous snapshot. Figure 6.8 illustrates the snapshot memory copy time for an unloaded and Application Runtime Microbenchmark-loaded guest. The no-load and Application Runtime Microbenchmark guest-loads both perform similiarly across the range of snapshot sizes. The memory footprint of the no-load and spin-load tests is very small. As a result, the memory copy time for delta-copy snapshot of various snapshot sizes is very small because only a limited number of pages will be dirtied. This test

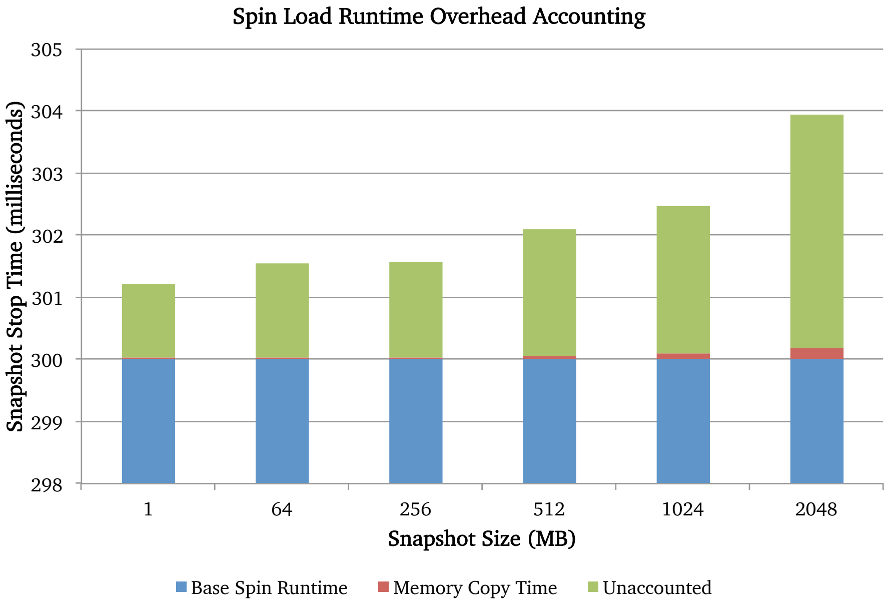


Figure 6.9: Accounting for Delta-Copy run time overhead in varying sized guests.

confirms the minimal memory impact of the Application Runtime Microbenchmark.

Now that the delta-copy snapshot mechanism under the Application Runtime Microbenchmark-load has been confirmed to perform similiary to when there is no guest load, the Application Runtime Microbenchmark can be used to create an accounting of the overhead of running the delta-copy snapshot. Figure 6.9 illustrates the run time overhead of delta-copy snapshots on a Application Runtime Microbenchmark-loaded guest being snapshotted at one Herz. The accounting is broken down into three parts: the base spin runtime, the baseline runtime of the Application Runtime Microbenchmark load; the memory copy time, the snapshot memory copy time directly measured by the guest; and, finally, the unaccounted stop time, which is the time leftover from the total guest load runtime less the base time and the memory copy time. As is expected, total overhead (memory-copy & unaccounted time) is reduced from the stop-copy accounting and especially the memory copy time has dropped to nearly zero. With delta-copy, the unaccounted time is now the vast majority of the performance impact. While not visible on this chart, the unaccounted time scales with the frequency of the snapshots, suggesting that much of the unaccounted overhead is contributed during the pause and restart phases of the stop-and-copy snapshotting while the guest is stopped and starting.

In addition to snapshot size, the effect of snapshot frequency on the delta-copy snapshot mechanism was investigated. Figure 6.10 illustrates the effect of snapshot frequency on the snapshot

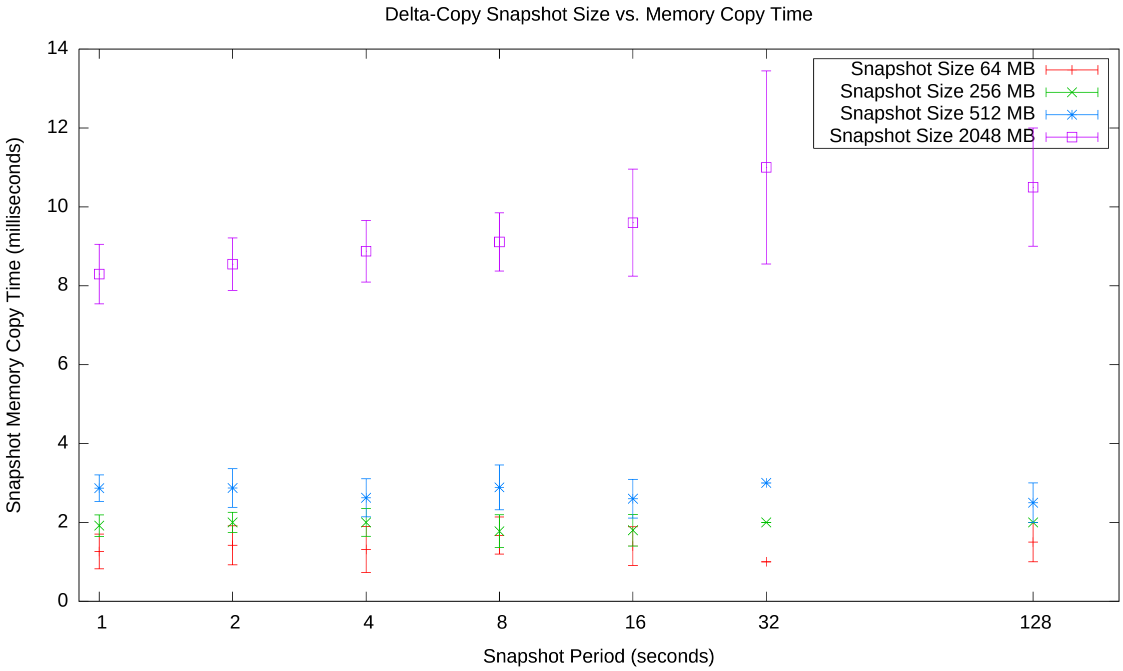
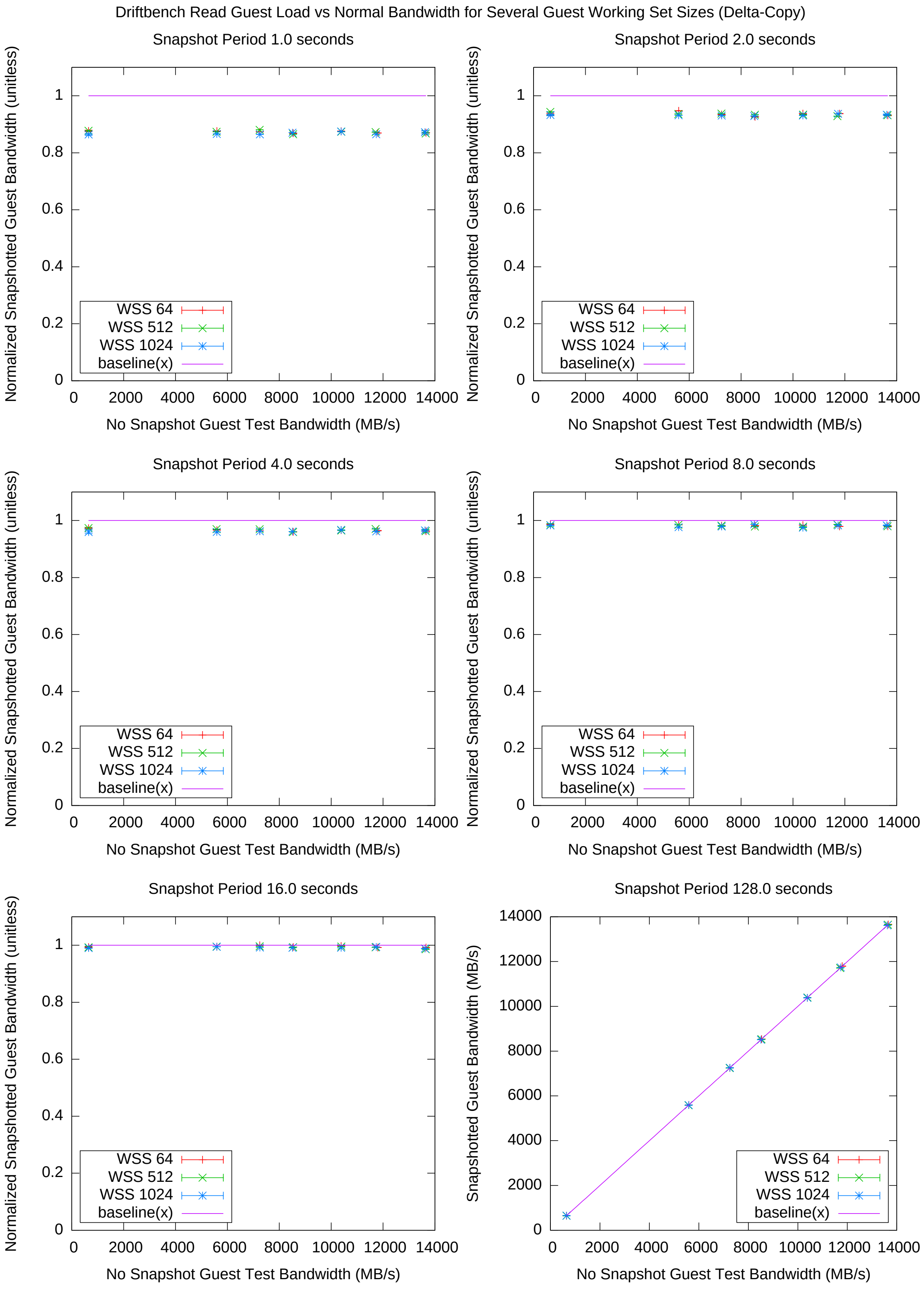


Figure 6.10: Effect of snapshot period on snapshot memory copy time for variously-sized unloaded guests.

memory copy time on an unloaded guest. Four snapshot sizes were investigated (64 MB, 256 MB, 512 MB, and 2048 MB) across a range of snapshot periods (1 s to 128.0 s). The average delta-copy snapshot copy times increased with the size of the snapshot but were very small. The delta-copy snapshot times averaged less than eleven milliseconds for all snapshot sizes (compared to hundreds of milliseconds for the stop-copy snapshots) with the larger snapshots taking longer than the smaller snapshots. Unlike with stop-copy snapshots, some frequency dependency was observed, with the 2048 MB snapshot averaging slightly longer copy times and higher variability at 128 seconds between snapshots than with one second between snapshots. This matches intuition because dirty pages due to operating system behaviors will increase with more time, causing increased dirty pages to be copied at snapshot time.

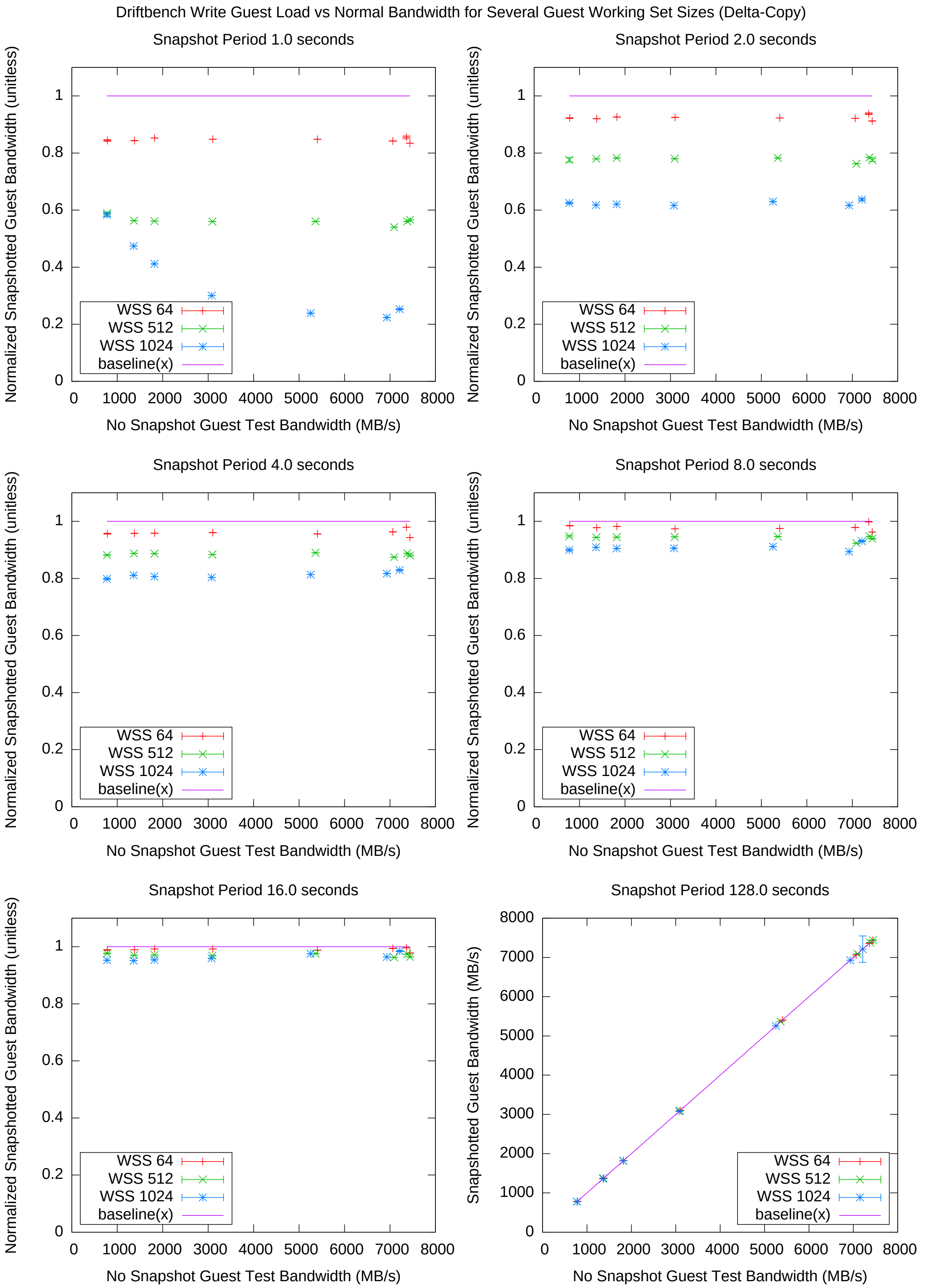
Delta-Copy: Guest Load

After evaluating the delta-copy on an unloaded guest, the delta-copy snapshotting mechanism is now examined in the context of a Memory Load Microbenchmark-loaded guest. Figure 6.11 illustrates runtime overhead of delta-copy snapshotting on a Memory Load Microbenchmark-loaded guest.



(a) Read Load

Figure 6.11: Runtime overhead of Delta-Copy Snapshotting on (a) read and (b) write guest loads with varying working set sizes and access rates. (Figure continues on next page.)



(b) Write Load

Figure 6.11: Runtime overhead of Delta-Copy Snapshotting on (a) read and (b) write guest loads with varying working set sizes and access rates.

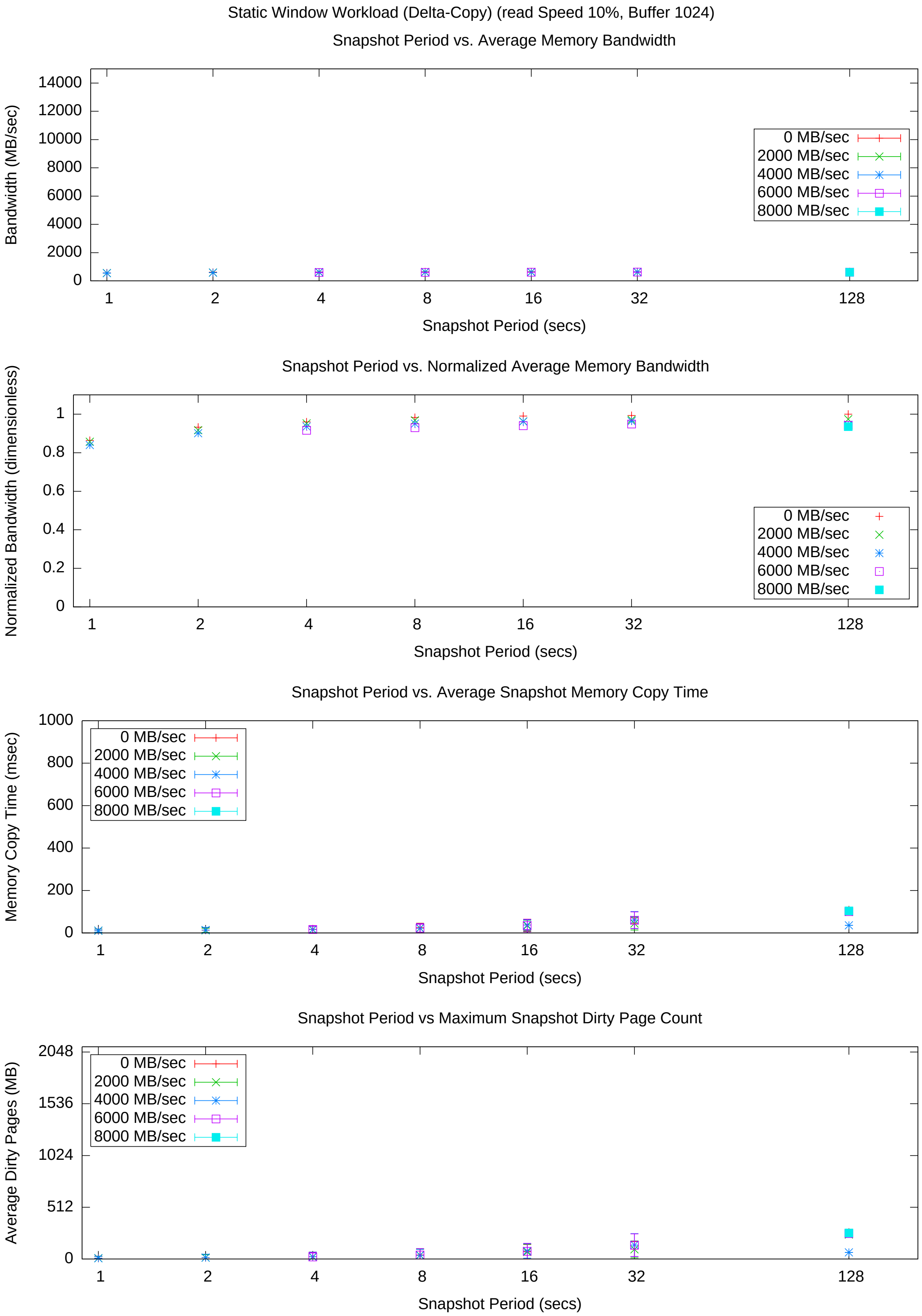
Subfigure 6.11(a) shows the results of testing a read load and subfigure 6.11(b) show the results of write load testing. Each subfigure contains six charts, each presenting the normalized access performance of Memory Load Microbenchmark in three working-set-size configurations (64, 512, and 1024 MB) across a range of access speeds. The 128.0 second snapshot period chart differs from the others because it presents the absolute performance of the benchmark against the baseline performance.

The baseline chart is changed in this way to illustrate that at 128.0 second snapshot period, the benchmark performs at baseline level. As the frequncy of delta-copy snapshotting increases, or the period decreases, the performance of the efficient introspection can be observed to decreate. For the read load, the normalized performance decrease is less than 20 percent at the one second snapshot period, and is flat across all access performance ranges tested. Further, the working-set-size of the guest read load does not effect the performance of the load under snapshotting.

In contrast to the read load, the normalized performance of the write loads does vary with working-set-size. At the two second snapshot period, the 64 MB WSS load slows to approximately 95% of the baseline, the 512 MB WSS to just less than 80%, and the 1024 MB WSS to nearly 60%. For the two second period these slowdowns are flat across access speed for all working-set-sizes, but for the one second period tests the slowdowns are write-speed dependent, with the slower write access rates showing better performance than the faster rates. At the shorter periods and slower access rates, their may not be time for the guest load to write the entire working-set buffer between snapshots, reducing the size of the dirty-page set to be copied, and reducing the performance impact of snapshotting. Both of these examples of a dependence between guest-load and the snapshotting performance reflect the underlying nature of the delta-copy snapshotting mechanism only copying dirty pages.

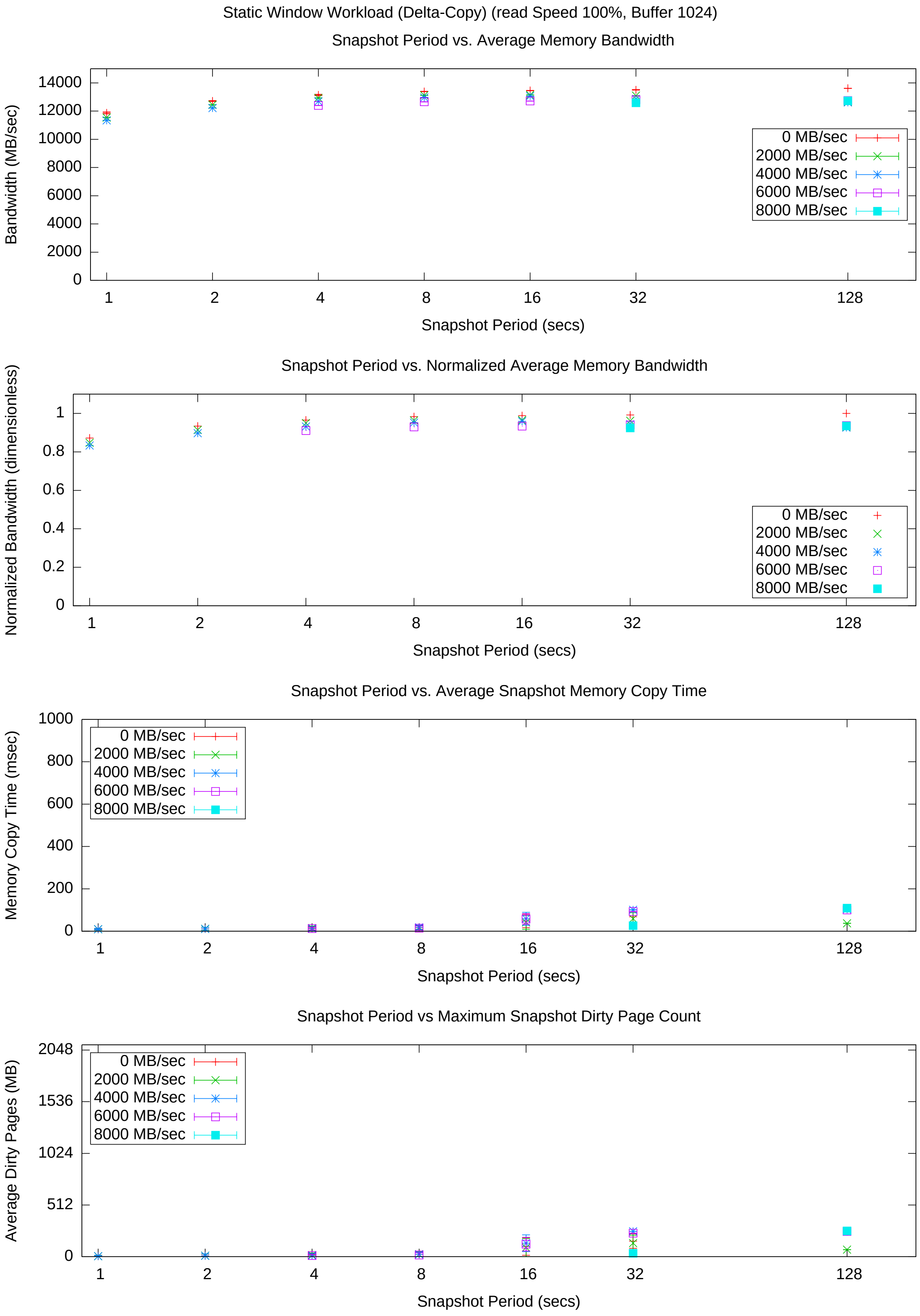
Delta-Copy: Introspection Load

After examining the effect of guest-loads on delta-copy snapshotting, introspection loads are now added to the snapshotted-guest load scenario. Figure 6.12 illustrates the runtime overhead of delta-copy snapshotting and introspection on a Memory Load Microbenchmark-loaded guest. The figure contains four subfigures that represent the four test series that were undertaken, Figure 6.7(a) shows a 10% baseline performance slow-read load, Figure 6.7(b) shows a 100% baseline



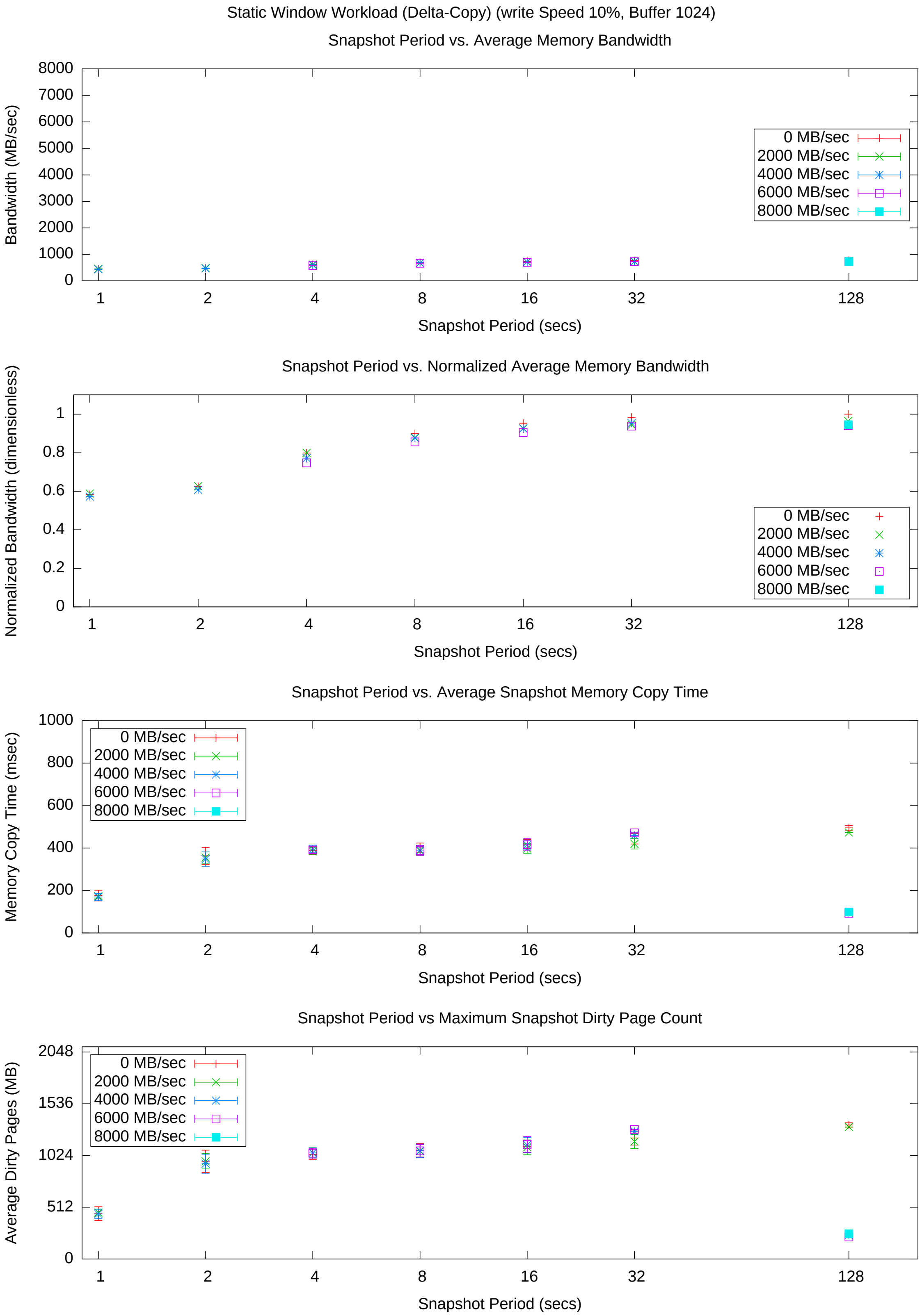
(a) Slow-Read Load

Figure 6.12: Runtime overhead of Delta-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with periods. Fast accesses performed at maximum rate and slow accesses rate limited to ten percent of maximum. (Figure continues on next page.)



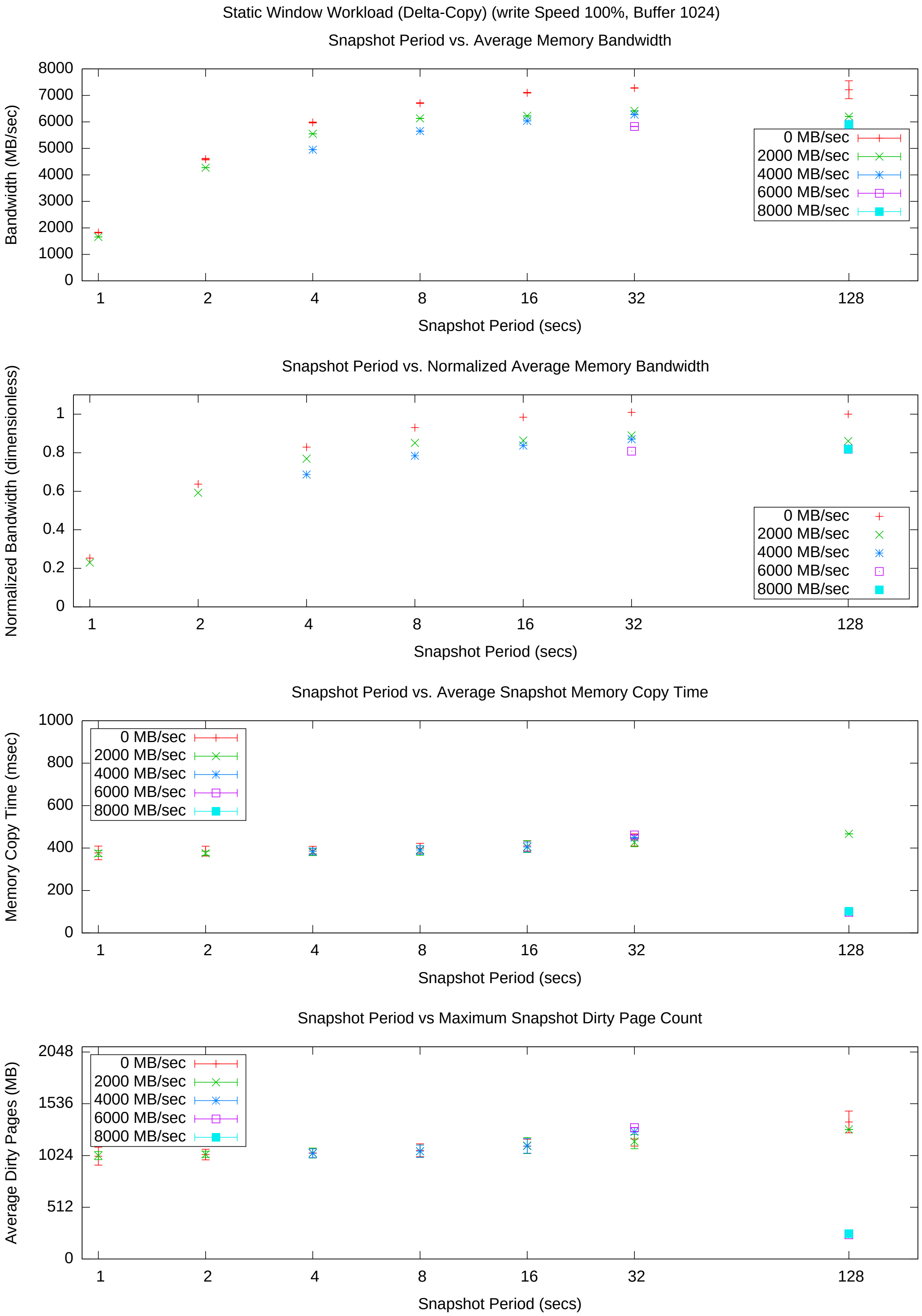
(b) Fast-Read Load

Figure 6.12: Runtime overhead of Delta-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with periods. Fast accesses performed at maximum rate and slow accesses rate limited to ten percent of maximum. (Figure continues on next page.)



(c) Slow-Write Load

Figure 6.12: Runtime overhead of Delta-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with periods. Fast accesses performed at maximum rate and slow accesses rate limited to ten percent of maximum. (Figure continues on next page.)



(d) Fast-Write Load

Figure 6.12: Runtime overhead of Delta-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with periods. Fast accesses performed at maximum rate and slow accesses rate limited to ten percent of maximum. (Figure continued from previous page.)

performance fast-read load, Figure 6.7(c) shows a 10% baseline performance slow-write load, and Figure 6.7(d) shows a 100% baseline performance fast-write load. Each of these subfigures contains four graphs: the top graph shows the absolute benchmark performance versus snapshot period, the second graph shows the normalized benchmark performance versus snapshot period, the third graph shows average snapshot memory copy time versus snapshot period, and the bottom graph shows the average dirty pages copied each snapshot versus snapshot period.

The test series measure the impact of simulated introspection by reading from the snapshot at varying rates (baseline 0 MB/s, 2000 MB/s, 4000 MB/s, 6000 MB/s, and 8000 MB/s) and were performed in the same manner as the stop-copy tests. Similar to stop-copy, the maximum observed delta-copy introspection rates decrease with snapshot period for all four test series. Different from stop-copy, the observed performance decrease is stronger with the fast-write load than was observed for slow-write or both read loads.

6.3.3 Drifting Load Evaluation

Three memory-intensive guest load applications, specifically Kernel Build, ClamAV Antivirus Scan, and Weka Machine Learning, were observed to incrementally operate on smaller sections of a larger dataset over time. This memory access pattern is referred to in this thesis as a "drifting memory window" access pattern. Delta-copy snapshotting displays an interesting behavior in the context of these types of loads that was hinted at in the delta-copy snapshot guest-load only evaluation for scenarios with short snapshot periods, slow-write access speeds, and large working-set-sizes. Tests performed under these conditions behaved in a manner similiar to a guest-load scenario with a smaller working-set-size because the load could not completely fill the working-set with new writes between snapshots.

Rather than a static window, as in the previous two subsections, the drifting window memory access pattern is now employed with the benchmark that allows the drifting window behavior to be observed over a larger range of guest load conditions. Figure 6.13 illustrates how varying access patterns can generate variously sized dirty page lists: pattern (a) writes into a 1024 MB static window, pattern (b) writes into two overlapping 512 MB drifting windows, pattern (c) writes into sixteen overlapping 64 MB drifting windows. Each of these patterns results in a unique Delta-Copy snapshot memory copy overhead. The sixteen 64 MB drifting window memory access pattern dirt-

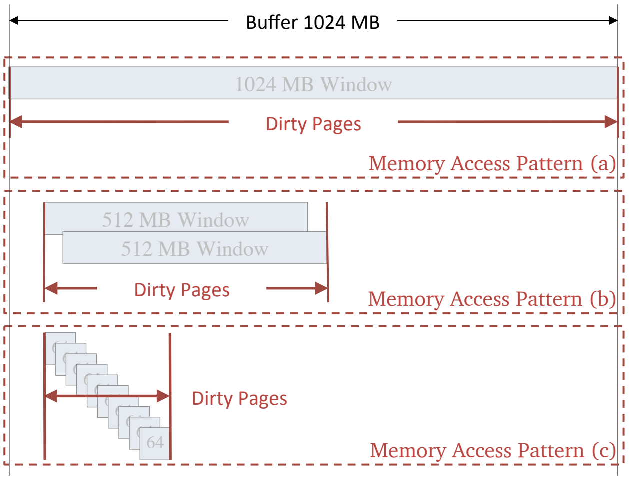


Figure 6.13: Memory access pattern comparison and the effect of various write patterns on dirty page creation. All three patterns write 1024 MB into the buffer but in different ways: pattern (a) writes 1024 MB into a static 1024 MB window, pattern (b) writes 1024 MB into two overlapping 512 MB drifting windows, pattern (c) writes 1024 MB total into sixteen overlapping 64 MB drifting windows. Each of these patterns results in different dirty page list sizes with corresponding effects on Delta-Copy snapshot memory copy overhead.

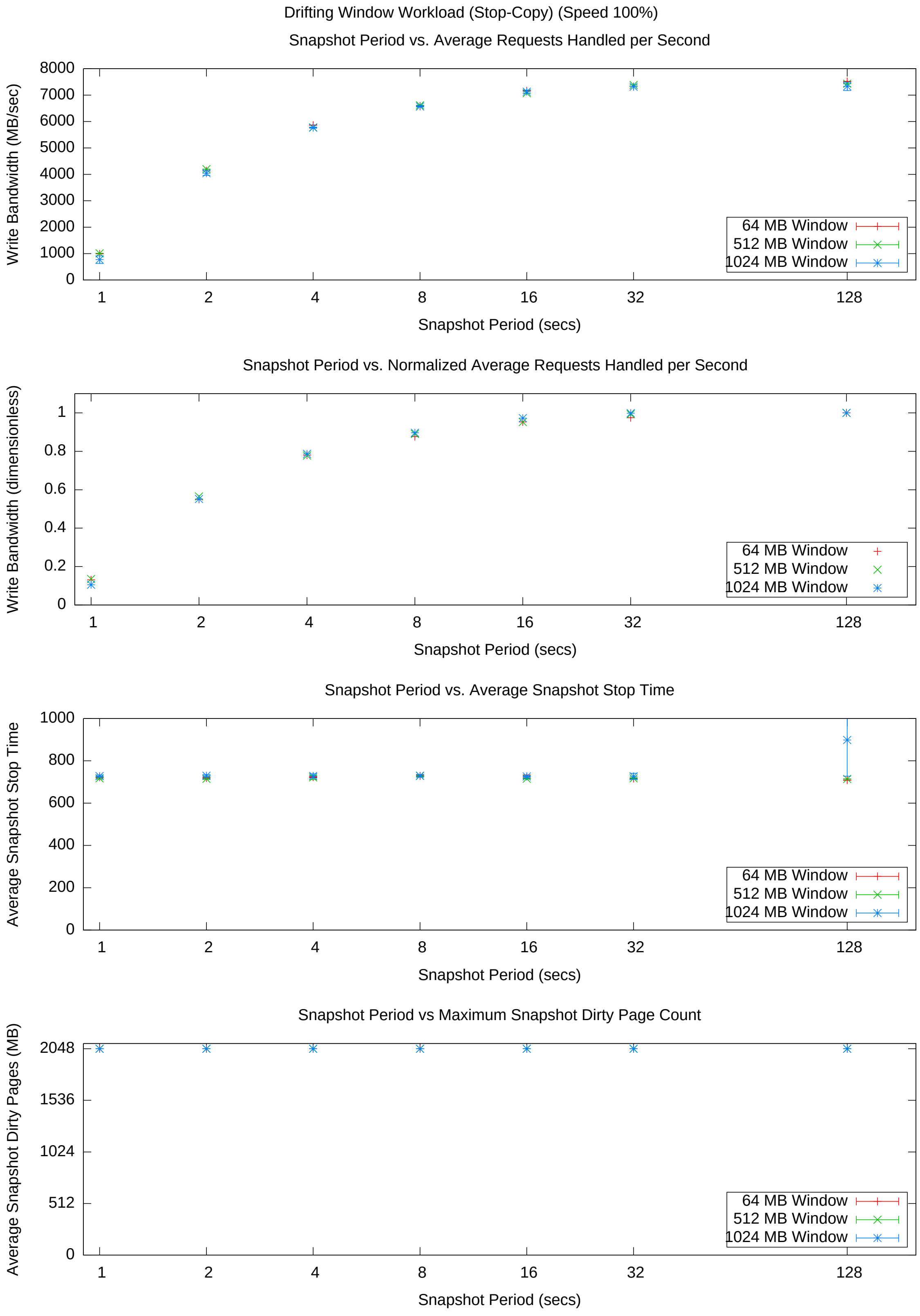
ies fewer pages than the single 1024 MB window, resulting in a faster delta-copy snapshot, despite the fact that each memory access pattern writes the same number of megabytes in each scenario.

Drifting-Load: No Load/Spin Load

There is no need to re-evaluate the no guest load conditions because only the guest load is changing; previous results for Delta-Copy Snapshot are sufficient.

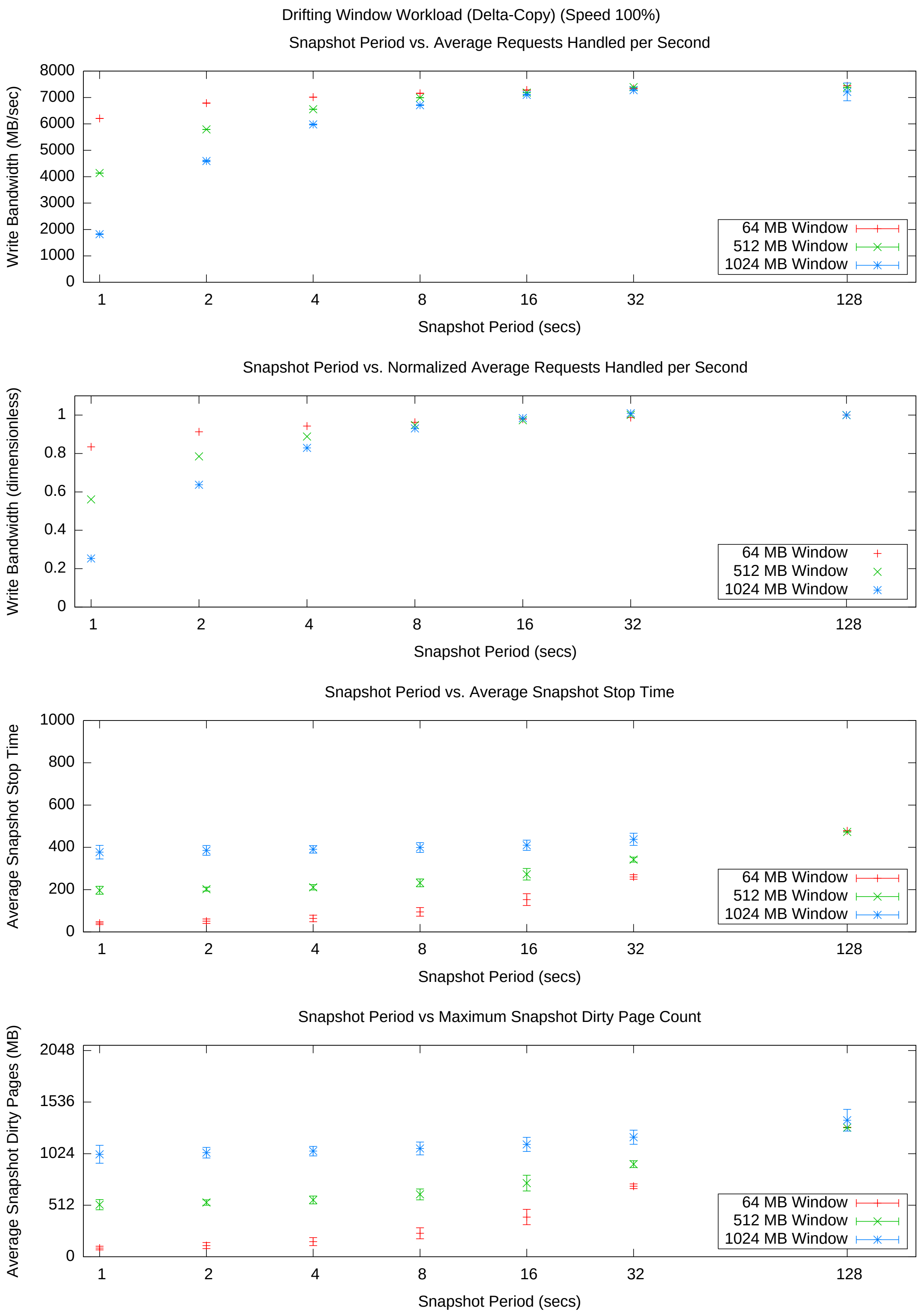
Drifting-Load: Guest Load

The snapshotting mechanisms are evaluated with drifting write loads in several conditions: three different drifting window sizes (64, 512, and 1024 MB), and snapshot periods (1-128 seconds) all writing at maximum rate 1024 MB buffers. Figure 6.14 illustrates how varying access patterns were observed to effect snapshotting overhead for both (a) Stop-Copy snapshots and (b) Delta-Copy snapshots. Each subfigure provides four different views into each testset, the top chart shows absolute memory bandwidth, the next chart shows normalized memory bandwidth, the second chart from the bottom shows average snapshot memory copy time, and the bottom chart shows average dirty pages copied per snapshot.



(a) Stop-Copy Snapshot

Figure 6.14: Effect of varying memory access patterns on snapshotting overhead for (a) stop-copy and (b) delta-copy snapshotted guests. (Continued on next page.)



Drifting Window Workload (Delta-Copy) (Speed 100%)

Snapshot Period vs. Average Requests Handled per Second

Snapshot Period vs. Normalized Average Requests Handled per Second

Snapshot Period vs. Average Snapshot Stop Time

Snapshot Period vs Maximum Snapshot Dirty Page Count

(b) Delta-Copy Snapshot

Figure 6.14: (Continued from previous page.) Effect of varying memory access patterns on snapshotting overhead for (a) stop-copy and (b) delta-copy snapshotted guests.

The Stop-Copy snapshot chart shows no difference between the working-set-sizes, which is expected because stop-copy snapshotting has no inherent guest-load dependency. As seen in the average dirty pages copied per snapshot chart, all pages are copied in each snapshot regardless of guest load behavior.

The Delta-Copy snapshot exhibits unique and interesting behavior with the drifting guest loads. As with the static-window loads tested in the previous section, the performance of the Memory Load Microbenchmark test changes with working-set-size; larger working sets perform worse because more pages must be copied per snapshot. However, the drifting-window guest load will cause the load to perform differently under different snapshotting period conditions. For example, at one second snapshot period the 64 MB drifting window can be observed to generate roughly 64 MB of dirty pages per snapshot but at 128.0 second snapshot period it has had time to dirty all 1024 MB of pages in it's buffer. In between these snapshot periods, the drifting window writes proportionally more memory as the period lengthens. As a comparison with the static-window tests of the previous section, the 64 MB drifting window in a 1024 MB buffer can be said to merge the performance of a 64 MB working-set-size static load at 1 second snapshot period and the 1024 MB working-set-size static load at the 128.0 second snapshot period.

Drifting-Load: Introspection Load

Introspection load testing of the drifting-load benchmark was not performed and may be explored in future work.

6.3.4 Pre-Copy Snapshot Evaluation

The Pre-Copy snapshotting mechanism is a variation on the Delta-Copy mechanism with the addition of a capability to eagerly copy memory pages into the snapshot before the snapshot stop time using a special precopy thread. Listing 6.2 contains pseudo-code describing the precopy thread that eagerly copies dirty pages after the snapshot has been released from introspection.

Listing 6.2: Precopy Thread Pseudo-Code Implementation

```
void precopy_thread()
{
    sync_dirty_pages();
    while( precopy_active ) {
        copy_if_dirty_and_clear( page++ );
        if( time > last_time + SYNC_PERIOD ) {
            sync_dirty_pages();
        }
    }
}
```

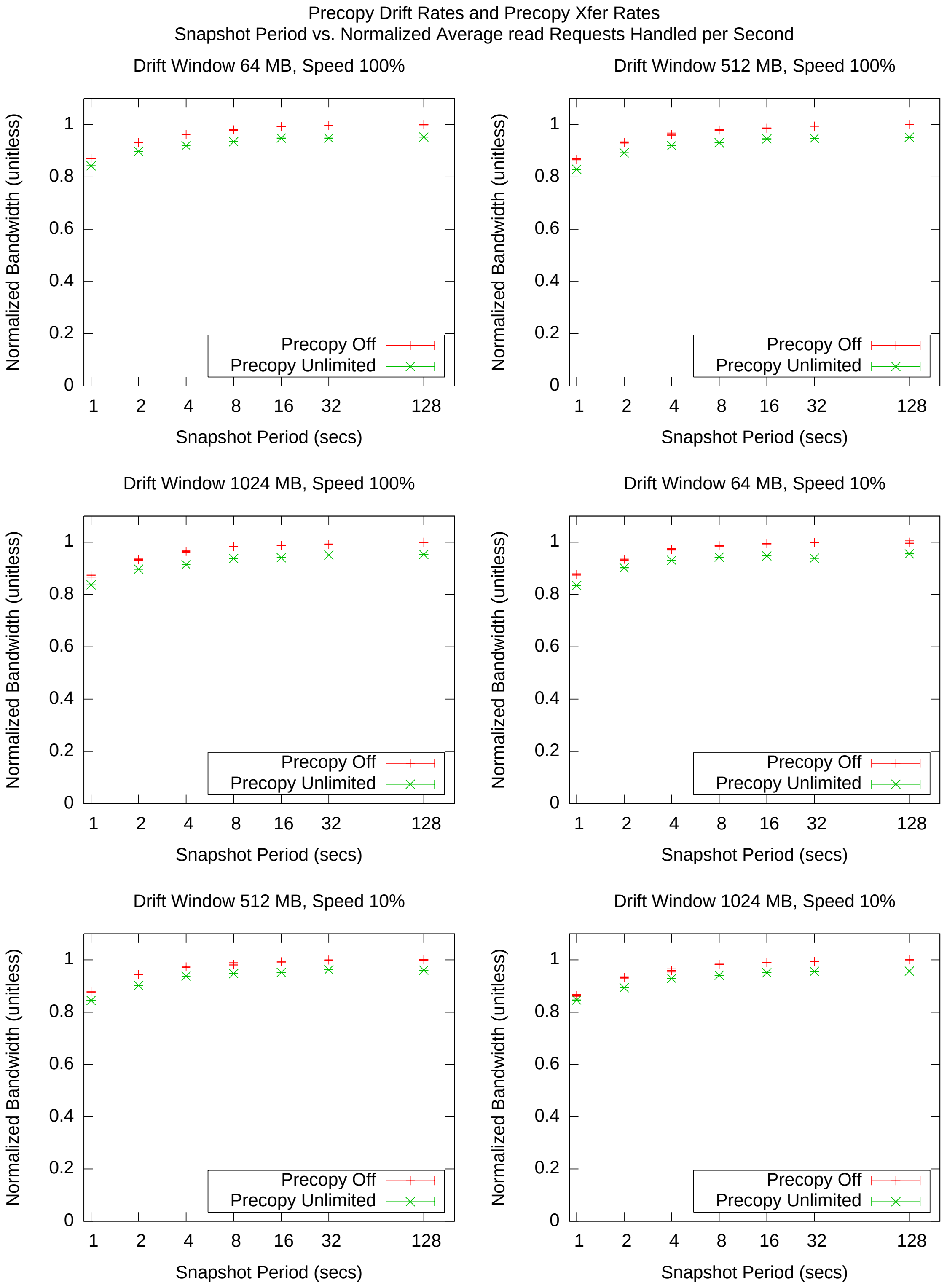
Pre-Copy: No Load/Spin Load

The No Load/Spin Load tests only validate the memory copy performance during snapshot time and so no results are presented for the Pre-Copy snapshot mechanism. This is because the precopy mechanism is only enabled after a snapshot has been released by the introspection application and is disabled before the snapshot is taken. The snapshotting mechanism for Pre-Copy is actually Delta-Copy, if the snapshot is never released by the introspection application.

Pre-Copy: Guest Load

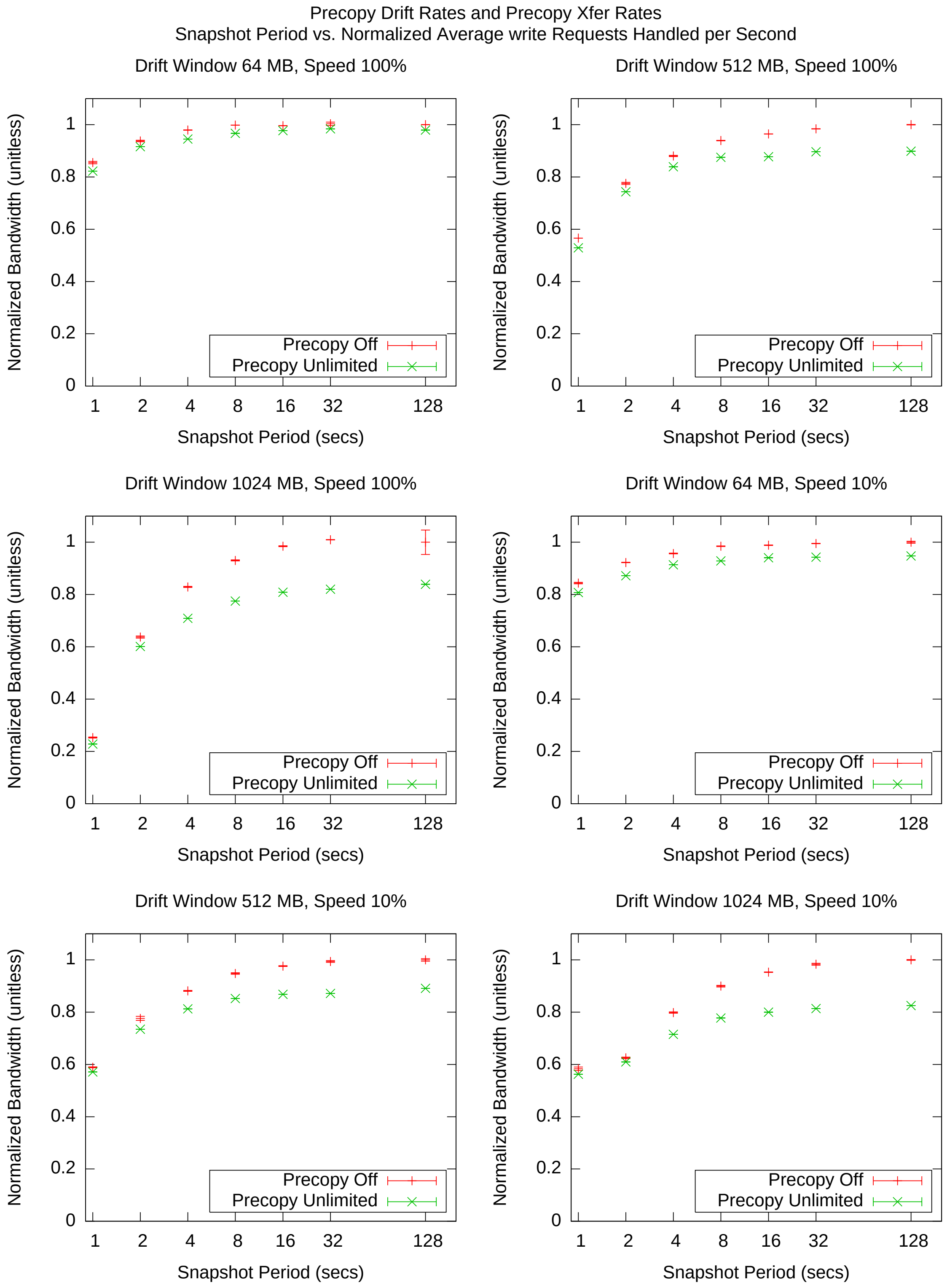
The evaluation of the Precopy mechanism was performed using a static window benchmark (read and write) of varying working-set-size (64,512,1024 MB) and access speed (10% and 100% of baseline maximum). Figure 6.15 contains two subfigures, (a) shows the reads, and (b) shows the writes. Each chart illustrates the normalized performance overhead of the microbenmark under test at varying snapshot periods with Precopy disabled (delta-copy) and Pre-Copy enabled with unlimited pre-copy transfer bandwidth.

In each case, the performance of the unlimited precopy condition is worse than the delta-copy. Pre-copy slows reads and writes alike, however, writes are most significantly impacted. The impact of pre-copy independent of snapshotting (the 128.0 s snapshot period), is 20% for both the 100% speed write and the 10% speed write. The microbenchmark performance decreases with buffer size for writes as the precopy thread increases. This effect is examined more in the key results, later on in this section.



(a) Read Load

Figure 6.15: Runtime overhead of Pre-Copy Snapshotting on (a) read and (b) write guest loads with varying working set sizes and access rates. Performance of delta copy (or ”Precopy Off”) is compared unlimited precopy rate performance. (Figure continues on next page.)



(b) Write Load

Figure 6.15: (Figure continued from previous page.) Runtime overhead of Pre-Copy Snapshotting on (a) read and (b) write guest loads with varying working set sizes and access rates. Performance of delta copy (or ”Precopy Off”) is compared unlimited precopy rate performance.

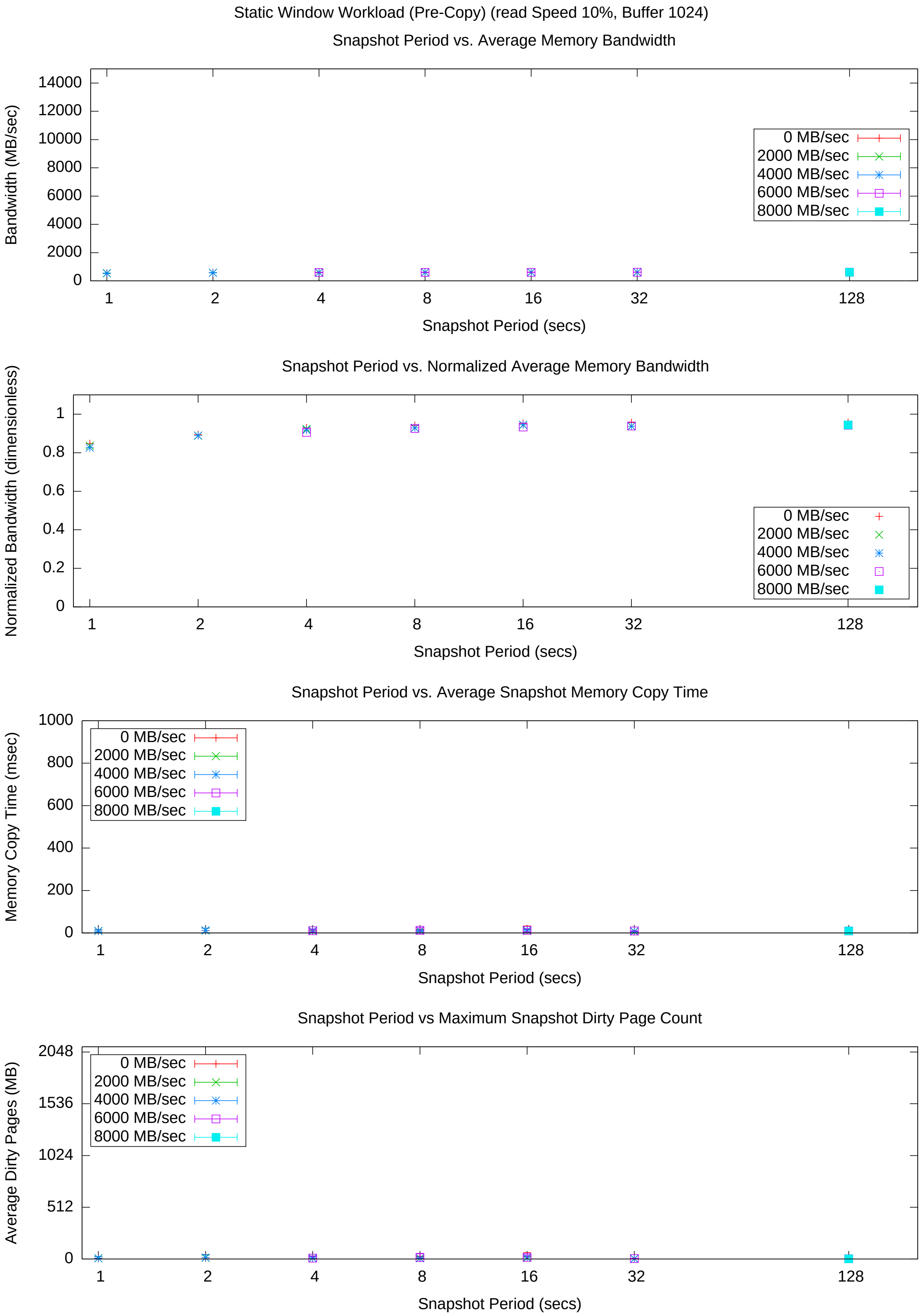
Pre-Copy: Introspection Load

After examining the effect of guest-loads on pre-copy snapshotting, introspection loads are now added to the snapshotted-guest with unlimited precopy scenario. Figure 6.16 illustrates the runtime overhead of the pre-copy snapshotting and introspection on a static-window Memory Load Microbenchmark-loaded guest. Figure 6.7(a) shows a 10% baseline performance slow-read load, Figure 6.7(b) shows a 100% baseline performance fast-read load, Figure 6.7(c) shows a 10% baseline performance slow-write load, and Figure 6.7(d) shows a 100% baseline performance fast-write load. Each of these subfigures contains four graphs: the top graph shows the absolute benchmark performance versus snapshot period, the second graph shows the normalized benchmark performance versus snapshot period, the third graph shows average snapshot memory copy time versus snapshot period, and the bottom graph shows the average dirty pages copied each snapshot versus snapshot period.

The introspection load test series measures the impact of introspection by reading from the snapshot at varying rates (baseline 0 MB/s, 2000 MB/s, 4000 MB/s, 6000 MB/s, and 8000 MB/s) and were performed in the same manner as the stop-copy and delta-copy tests. Similar to stop-copy, the maximum observed delta-copy introspection rates decrease with snapshot period for all four test series. Different from stop-copy, the observed introspection performance decrease is stronger with the fast-write load than was observed for slow-write or both read loads.

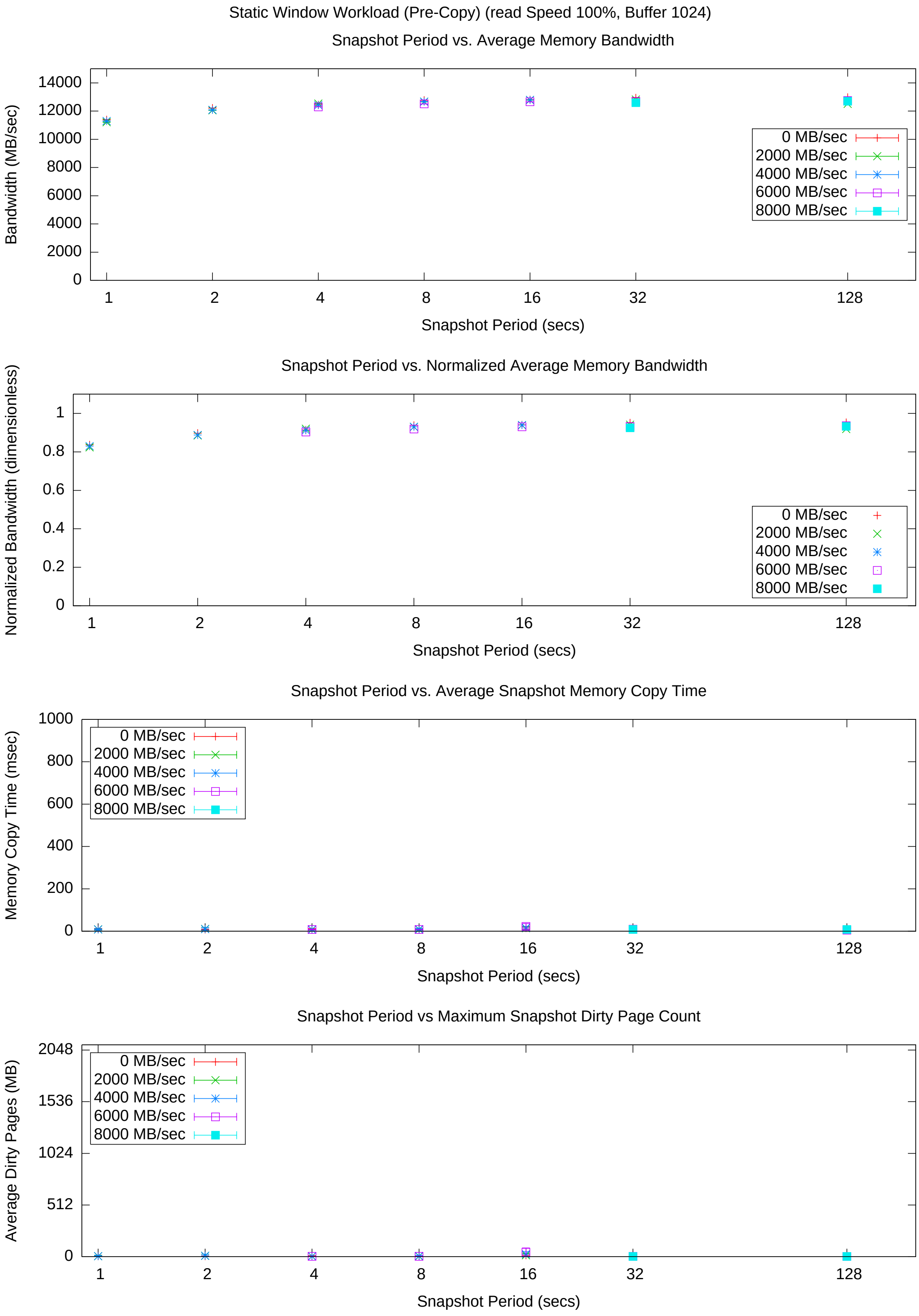
The Pre-Copy snapshot with introspection results provide a unique view into the behavior of the pre-copy mechanism. The average of dirty page count is observed to fall to nearly zero for both of the 128 second snapshot period write tests and decrease slightly in the 32.0 period tests. This result implies that the pre-copy mechanism is functional, even if it is not copying enough pages to overcome it's overhead by reducing the snapshot copy costs.

More research will be required to understand why the pre-copy mechanism does not increase performance. The mechanism may be useful in regimes that were not explored here. Further research into reducing the *unallocated* snapshot stop-time costs unrelated to memory copying may change the balance of costs that contributes to the failure of the Pre-Copy snapshotting mechanism.



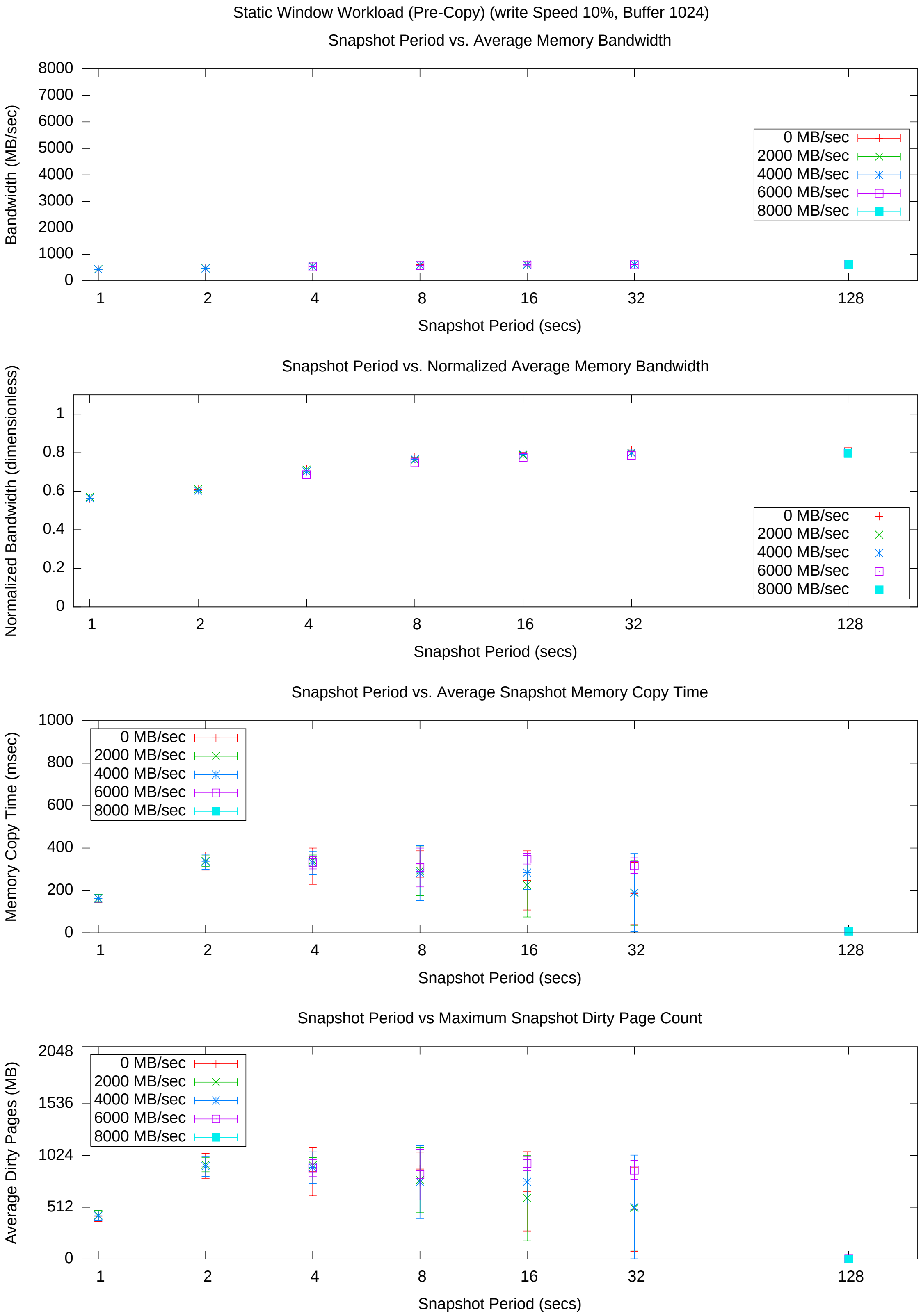
(a) Slow-Read Load

Figure 6.16: Pre-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with varying periods. Fast accesses at max rate and slow accesses rate limited to ten percent of max. Pre-Copy rate unlimited for all tests shown. (Figure continues on next page.)



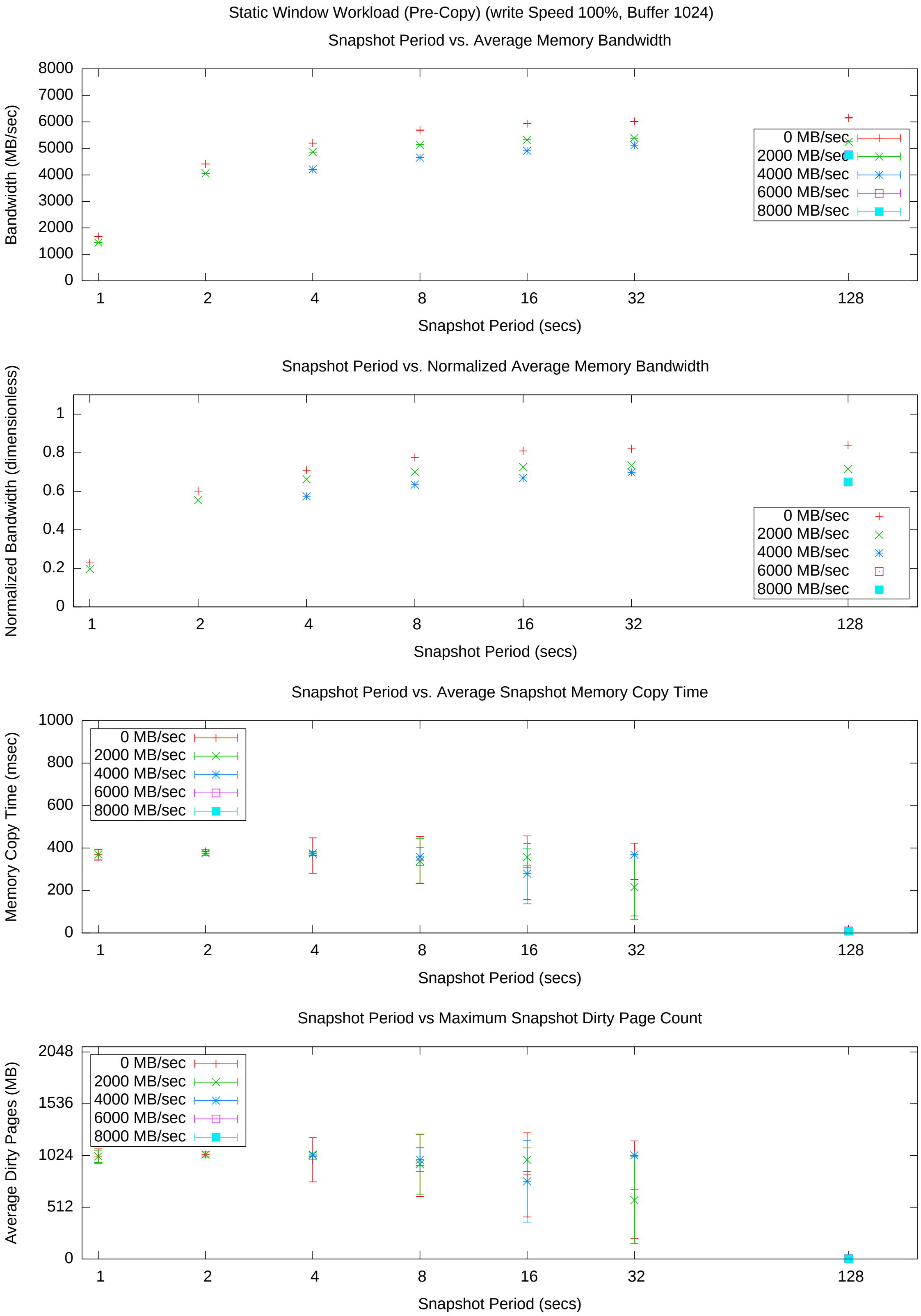
(b) Fast-Read Load

Figure 6.16: Pre-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with varying periods. Fast accesses at max rate and slow accesses rate limited to ten percent of max. Pre-Copy rate unlimited for all tests shown. (Figure continued on next page.)



(c) Slow-Write Load

Figure 6.16: Pre-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with varying periods. Fast accesses at max rate and slow accesses rate limited to ten percent of max. Pre-Copy rate unlimited for all tests shown. (Figure continues on next page.)



(d) Fast-Write Load

Figure 6.16: (Figure continued from previous page.) Pre-Copy Snapshotting on (a) slow-read, (b) fast-read, (c) slow-write, and (d) fast-write guest loads with varying periods. Fast accesses at max rate and slow rate limited to ten percent of max. Pre-Copy rate unlimited for all tests shown.

6.4 Microbenchmark Evaluation: Key Results

The systematic evaluation of the microbnechmark introspection scenarios revealed several key results about efficient introspection. This section will review the microbenchmark evaluation, call out those key results, and provide some discussion.

6.4.1 Snapshot Frequency Most Significant Influence on Guest Performance

Snapshot frequency dictats how often the guest is snapshotted. In order ot maintain coherence, the snapshots are performed with the guest paused. While the length of these pauses are be dictated by the specific properties of the snapshotting mechanism and guest load, the cost of pausing can be amortized by performing fewer snapshots in a given period of time.

6.4.2 Delta-Copy Snapshot Offers Superior Performance

Delta-Copy snapshotting offers performance gains for guest loads and is the best performing snapshot solution. A wide variety of memory access patterns were examined in the microbenchmark evaluation but delta-copy snapshotting offered superior performance over stop-copy and pre-copy. Delta-copy snapshotting is the preferred snapshotting mechanism for normal guest operation.

6.4.3 Unaccounted Snapshot Stop-Time

The actual causes of slowdown due to the unaccounted snapshot stop-time are not fully understood. In addition to the snapshot memory copy time, which is directly measurable, the *unaccounted* slowdown accounted for Stop-Copy in Figure 6.4 and Delta-Copy in Figure 6.9 is a significant factor in snapshot stop time. The unaccounted snapshot stop-time was so significant that it dictated the minimum snapshot periods tested in this work.

Further research should explore whether these slowdowns are attributable to the process of pausing and restarting the guest. If the unaccounted slowdown is related to pause-restart, large performance gains could be recovered by modifying the hypervisor pause-and-resume mechanisms to improve efficiency of snapshotting while still maintaining memory coherency.

6.4.4 Dirty Page Tracking is Cheap

Delta-Copy snapshotting relies on a dirty-page tracking mechanism that is implemented by the KVM kernel driver. Table 6.2 compares guest Memory Bandwidth Microbenchmark read and write performance with dirty page tracking enabled and disabled for a range of buffer sizes. The performance impact of dirty page tracking is observed to be negligible for both reads and writes over a range of buffer sizes. Dirty page tracking is used throughout virtualization and has been engineered to be cheap.

6.4.5 Introspection Impact on Guest

The systematic evaluation of the microbenchmark introspection scenarios revealed that introspection does not impact snapshotting stop time, but can impact guest-load performance through memory bandwidth competition. Figure 6.17 illustrates impact of introspection on the guest isolated from snapshotting. The write guest load is observed to approach a bandwidth cap of 6000 MB/s in the presence of an 8000 MB/s introspection read load. The read guest load is observed to suffer a uniform performance impact across all guest loads when in the presence of introspection loads. The exact underlying causes of these impacts is left unexplored but likely require detailed, platform specific architectural-level modeling to explain. In a practical virtualized system – like the systems assumed to be present in this thesis – the guest-load would not only be competeing with the introspection for memory bandwidth, but also with other guests and their potential introspection, limiting the utility of such models in predicting practical system performance.

lmbench	Read Test (MB/s)		Write Test (MB/s)	
Buffer Size (MB)	No Tracking	With Tracking	No Tracking	With Tracking
1 MB	7135	7165	10387	10231
64 MB	5936	5938	7885	7674
1024 MB	5908	5957	7896	7808
2048 MB	207	183	233	247

Table 6.2: Memory Bandwidth Microbenchmark performance with dirty page tracking enabled and disabled for a range of configurations.

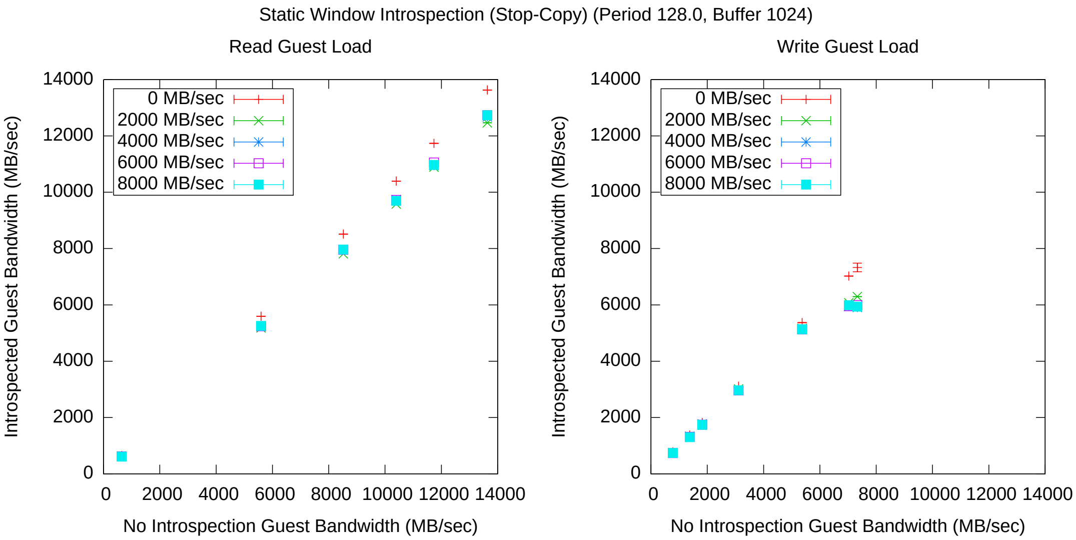


Figure 6.17: Impact of varied introspection loads on the Memory Load Microbenchmark isolated from snapshotting.

6.4.6 Stop-Copy Snapshotting Impacts Only Guest Runtime

The performance impact imposed upon guest loads by the Stop-and-Copy snapshot mechanism is observed to be independent of the three guest loads tested: no load, Application Runtime Microbenchmark, Memory Load Microbenchmark. Figure 6.3 shows that the snapshot memory copy time is not affected by the Application Runtime Microbenchmark compared to an unloaded guest. The overheads accounted for the Application Runtime Microbenchmark-loaded guest in Figure 6.4 correspond with with the overheads observed on the Memory Load Microbenchmark-loaded guest configuration illustrated in Figure 6.6. Stop-and-Copy snapshotting is expensive in terms of snapshot-stop time overhead compared to Delta-Copy but offers a very consistent stop time across all observed guest-loads, which could be useful for some introspection application scenarios.

6.4.7 Dirty Page List Synchronization is Expensive

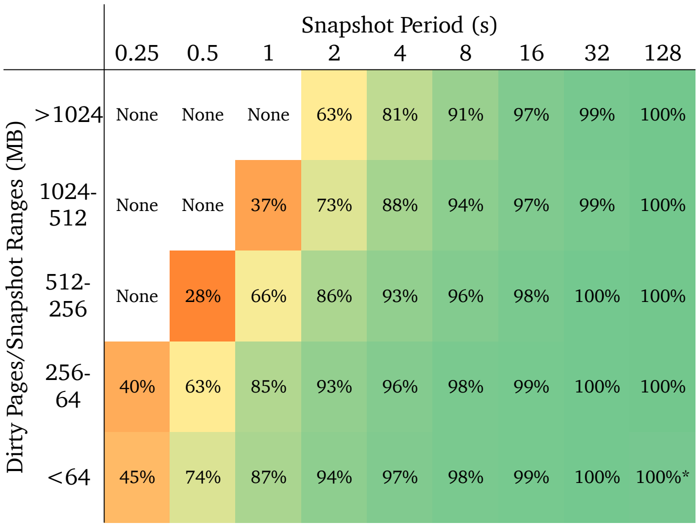
Dirty page tracking is cheap, but synchronizing the list of dirty pages between the tracking mechanism and the efficient introspection is expensive. Table 6.3 compares Memory Bandwidth Microbenchmark read and write performance with dirty page table synchronization performed at various frequencies. The read performance of the guest is largely unaffected by synchronization even when synchronization is performed four times per second, but guest write performance is halved at 2 Hz synchronization frequency with a 1024 MB guest write buffer. This is not a problem for the Delta-Copy snapshotting mechanism, as it must only synchronize the dirty page tracking once per snapshot, that synchronization happens while the guest is paused, and the costs of tracking are small compared to the impact of the snapshot itself. The Pre-Copy snapshotting mechanism relies on searching the dirty-page list for new pages to eagerly pre-copy while the guest is still running. Updateing the dirty-page list more frequently during guest operation reveals more pages to pre-copy. The expense of dirty-page list synchronization limits the frequency that new pages can be pre-copied and hamstrings the effectiveness of the pre-copy snapshot mechanism overall.

6.5 Efficient Introspection Performance Guidance

The systematic evaluation of the efficient introspection with microbenchmarking revealed several key facts that will provide guidance on new applications. First, the frequency of snapshotting and the memory access pattern of the guest load are determining factors in predicting an applications performance under introspection. Second, Delta-Copy snapshotting reduces performance overhead compared to Stop-Copy and Pre-Copy snapshotting. Figure 6.18 is a heat map of observed performance over all the Delta-Copy Memory Load Microbenchmark trials presented in this thesis regardless of specific configuration. The results were aggregated by binning the trials by snapshot period (0.25,0.5,1,2,4,8,16,32, and 128 seconds) and observed average dirty pages per snapshot ranges (>1024, 1024-512, 512-256, 256-64, and <64 MB). Each bin is annotated with the average normalized performance overhead and colored to indicate the amount of impact; 100% performance

lmbench	Read Test (MB/s)		Write Test (MB/s)			
Buffer Size (MB)	No Sync	4 Hz	No Sync	1 Hz	2 Hz	4 Hz
1 MB	7135	6830	10387	9916	9903	9896
64 MB	5936	5657	7885	7610	7841	7705
512 MB	5977	5732	7773	7845	7816	**3654**
1024 MB	5908	5715	7896	7771	**3734**	**2611**
2048 MB	207	152	233	171	192	188

Table 6.3: Memory Bandwidth Microbenchmark (lmbench) performance with dirty page synchronization performed with various frequencies. Only dirty page synchronization was performed, no memory was copied. The highlighted figures indicate an observed performance impact at 4 Hz for the 512 MB lmbench write and 2/4 Hz for the 1024 MB lmbench write.



Dirty Pages/Snapshot Ranges (MB)	Snapshot Period (s) 0.25	0.5	1	2	4	8	16	32	128
>1024	None	None	None	63%	81%	91%	97%	99%	100%
1024-512	None	None	37%	73%	88%	94%	97%	99%	100%
512-256	None	28%	66%	86%	93%	96%	98%	100%	100%
256-64	40%	63%	85%	93%	96%	98%	99%	100%	100%
<64	45%	74%	87%	94%	97%	98%	99%	100%	100%*

Figure 6.18: Efficient introspection delta-copy snapshot performance heat map for all tests presented in this thesis. *Note: no tests were observed with snapshot period 128.0 and less than 64 MB of dirty pages but performance in this regime will be 100%.

means no impact and is colored green, decreasing performance is more red. The general trends of the heat map are that the longer snapshot period bins how better Memory Load Microbenchmark performance and that the performance of microbenchmark increases as the average dirty pages per snapshot decreases. No tests were observed to complete in the low-period high-dirty-pages corner of the heat map. Those bins are labeled ”none” and colored white. It should be noted that no trials were observed with less than 64 MB of average dirty pages at the 128.0 snapshot period, likely because of operating system memory overheads unrelated to the loads, but intuition suggests that performance should remain 100% that scenario.

The performance heat map can provide guidance in the prediction of a potential guest loads performance with efficient introspection. The table would have to be computed for application to a specific platform. In this way, the introspection application developer could tune the snapshotting period to the memory access pattern of their guest load and find a performance overhead level that met up with their definition of ”normal guest performance.” These predictions will be applied to several potential introspection application scenarios in the next chapter.

Chapter 7

Potential Applications

This chapter discusses issues surrounding the implementation of two potential introspection security applications. These applications are the signature-based antivirus scanner, which demonstrates full memory signature generation at a single moment in time, and the network integrity manager, where packets passing the guest-based firewall are verified against packets routed by the host. These two security applications were previously too slow to tackle without efficient introspection. The limitations associated with previously-existing introspection plaforms are presented in comparison.

7.1 Introspection Application Performance Goals

This thesis defines the introspection application performance target as follows: for a given task, introspection should incur no additional penalty over performing that same task in a non-introspected environment. For example, if a guest were performing a task while simultaneously sweeping memory for the presence of a virus, then the time to perform the sweep and the time to perform the task should not vary when the memory sweep is performed from the hypervisor using introspection instead of in the guest. Another important goal of performance evaluation will be characterising the performance relative to existing introspection patterns (e.g. pause-resume, small atomic checks, and incoherent memory sharing). The exact choice of performance metrics and guest workloads will be chosen to match the specific scanning technique being demonstrated.

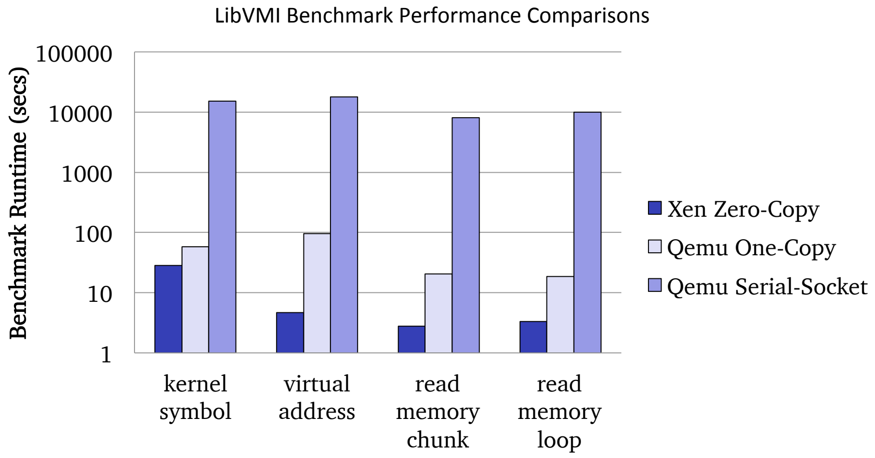


Figure 7.1: LibVMI benchmarks (kernel symbol translation, virtual address translation, read memory chunks, and read memory byte-by-byte) comparing performance between three interfaces: Xen Zero-Copy, KVM/QEMU One-Copy Socket, and KVM/QEMU Serial Socket.

7.2 Potential Application: Antivirus Signature Memory Scan

Signature-based antivirus identifies virus infected memory by comparing the checksums of memory against a list of the checksums of known virus infected memory. Various hashing techniques have been used for memory signature generation including MD5 hashes for Copilot by Petroni et al. [16], SHA-1 for SecVisor by Seshadri et al. [17], or custom signature generation schemes like that developed by Dolan-Gavitt et al. [18].

7.2.1 Previous Antivirus Capability

Previous introspection platforms that did not meet the requirements for Efficient Introspection were evaluated for implementation of memory-signature antivirus scanning and found to be inadequate. Figure 7.1 shows introspection memory access performance measurements for LibVMI accessing a Xen guest, LibVMI accessing a KVM guest with a socket based interface, and LIbVMI accessing a KVM guest with a serial style interface. While this figure only compares the relative runtimes of the LibVMI benchmarks, the read access performance of the Xen guest was observed to operate at the native memory access of the host computer.

Xen can be observed to access guest memory at native memory access speeds and at first glance appears to be a competitive platform for the implementation of signature-based antivirus introspection. However, in order to realize the benefit of memory coherency, the Xen guest would have to

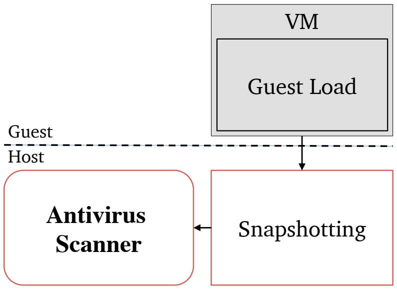


Figure 7.2: Block diagram showing the host-based Antivirus software performing memory hashing on the introspected guest.

be paused for the entire antivirus scan. The Xen guest could also be run in parallel with the scan, removing the guest-stop overhead entirely, but then a clever virus might be able to hide it's presence by moving through memory to stay ahead and out of reach of the signature generation mechanism. Only scanning the entire memory at a single point in time can guarantee the signature mechanism access to malicious memory.

The LibVMI interfaces to KVM can both be seen to operate at significantly lower than native memory performances, approximately 5x slower for LibVMI One-Copy and 3000x slower for LibVMI serial. These inefficient interfaces are themselves a bottleneck to implementation of a memory intensive introspection application like memory-signature scanning without efficient introspection.

7.2.2 Antivirus with Efficient Introspection

The goal of the Antivirus Signature Memory Scan demonstration is not to create a practical virus detection tool, instead, the goal is to define a limit to the memory bandwidth available to the introspection check, exercise the introspection memory system, and demonstrate the feasibility of a previously performance-limited application. Figure 7.2 shows a block diagram describing the process of the host-based Antivirus Software performing memory hashing on the introspected guest. The VM guest running it's guest load is snapshotted by the hypervisor when triggered by the Antivirus Scanner introspection application. The hypervisor then shares the snapshotted memory with the Antivirus scanner.

To this end, complex signature hashes will be eschewed in favor of minimizing CPU calculation. Memory bandwidth requirements will be maximized by calculating a simple XOR-based checksum for each page in guest physical memory for a given snapshot as shown in Listing 7.1. In the case

Listing 7.1: Antivirus Case Study memory page XOR-hashing algorithm.

```

uint64_t HashThisPage( uint64_t *memory_page, long page_size_bytes )
{
    uint64_t hash = 0;
    int i = 0;

    for( i = 0; i < (page_size_bytes / sizeof(uint64_t)); i++ ) {
        hash = hash ^ memory_page[i];
    }
    return hash;
}
```

of the computer used for this test, XOR-hashing was performed at approximately 8 GB/s. This compares favorably with the maximum read speed observed on the same computer of 8.2 GB/s, suggesting that the XOR-hashing algorithm was essentially memory bandwidth limited. The time to create a complete page-by-page checksum list of the entire guest memory was recorded as the time to perform a complete scan. Pause-resume models used previously would lose interactivity by stopping the VM guest long enough to perform a complete scan.

7.2.3 Performance Evaluation of Antivirus with Efficient Introspection

The efficient introspection microbenchmark performance heat map from the previous chapter has been overlayed with Antivirus-specific restrictions in Figure 7.3 contains The 0.25 second snapshot period position on the performance heat map have been eliminated because the introspection application would not have time to complete the antivirus signature generation between each of the snapshots. At 0.5 seconds or above, the impact of the antivirus scan will decrease with increasing between snapshots and checks. For given guest loads, the impact of snapshotting will decrease with the dirty page access footprint of the guest load. The efficient introspection performance estimate heat map can provide guidance on the implementation of the the Antivirus Signature Memory Scan.

Limitations of Antivirus Signature Memory Scan

The signature-based antivirus scanner only mimics the memory scanning action of a traditional signature-based antivirus memory scanner and does not diagnose actual virus infections against

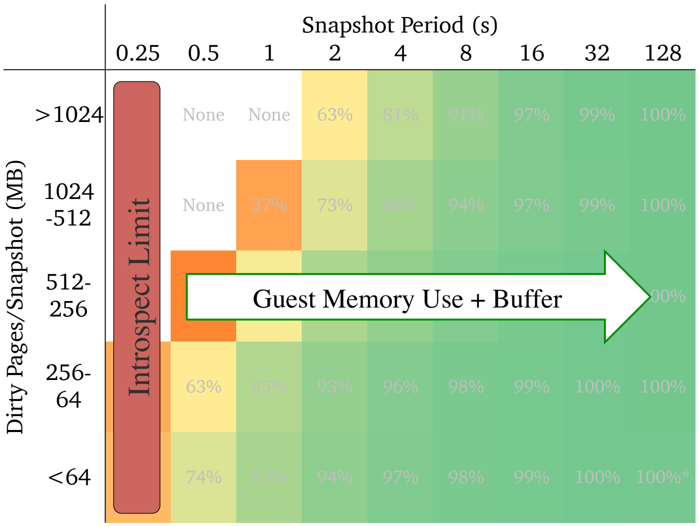


Figure 7.3: Efficient introspection microbenchmark performance heat map overlayed with Antivirus specific limitations.

a large corpus of signatures. The performance profile of the Antivirus Signature Memory Scan demonstration application will differ from that of a genuine Antivirus Signature Scanning software in terms of memory, CPU, and disk use.

7.3 Potential Application: Network Integrity Manager

The Network Integrity Manager identifies virus infected guests by verifying packets allowed through the guest-based firewalls against packets observed by the host. Packet verification reveals the presence of rootkits in the guest that, like the Mebroot rootkit, evade guest-based application firewalls by injecting packets into undocumented interfaces within the NDIS netowrk driver stack.

7.3.1 Previous Network Scanning Capability.

Previously, with VMware VProbes, memory introspection performance issues limited the verification to simply counting packets. VProbes operates by running a call-back routine that introspects the guest when certain triggering events occur. In the case of the NetIM network scanner, the triggering event was the guest network firewall passing a packet through to the network. The introspection action of the initial NetIM prototype was to count that a packet had been passed by the guest firewall.

Copying any properties of the packet out of memory was not possible due to the nature of the

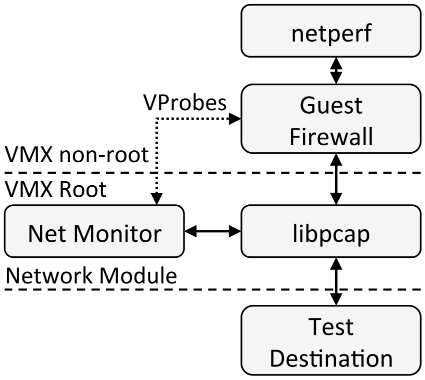


Figure 7.4: This block diagram describes the major features of the Network Integrity Module and it's associated testing framework. The Net Monitor compares the traces collected with vprobes and libcap. Netperf generates defined network traffic to the test destination.

VProbes introspection implementation. The VMware hypervisor pauses the guest while running the introspection call-back routine. VMware VProbes actively limits the runtime of introspection call-back routines to prevent the VProbes from incurring an adverse impact on the guest.

Currently the limiting factor in the performance of the Network Integrity Manager is the slowdown imposed by instrumenting the guest firewall with the VMWare VProbes. For these tests the Ubuntu 8.10 Intel Core-i7 host with 6 GB of RAM is running the VMWare Workstation 7.1.15 hypervisor deploying a Windows XPSP1 guest with 1 CPU and 512 MB of RAM. The network performance was evaluated with netperf 2.4.5 [19] built to support spin waits between packet bursts and running a standard TCP_STREAM test. The baseline system performed the netperf TCP test at 312.93 Mbits/second. Enabling only the VProbes instrumentation with no data logging decreased the netperf TCP test performance 37.35% to 196.04 Mbits/second. The counting-only Network Integrity Manager reduced performance 42.11% to 181.16 Mbits/second.

Despite these limitations, the counting-only network packet verification can identify instances of guest infection by the Mebroot virus. The Network Monitor compares both the host- and guest-based network traffic views and Figure 7.4 is a block diagram of system.

While count-only verification can identify packets injected by the Mebroot sample, the efficient introspection memory access could enable more advanced checking, like full packet comparison, that could detect packet redirection man-in-the-middle attacks (and others).

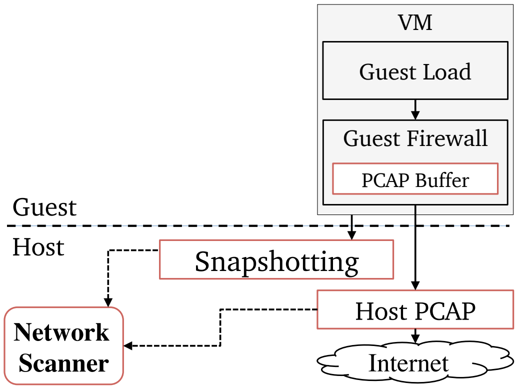


Figure 7.5: Block diagram showing the host-based NetIM software performing differential analysis comparing the outgoing packets passed by the guest firewall with the packets observed leaving the guest.

7.3.2 NetIM with Efficient Introspection

In the initial VProbes implementation of the Network Integrity Manager, introspection performance issues limited the implementation to only counting packets passed. Better security could be provided if the NetIM could provide more complete packet comparisons. Efficient introspection potentially provides the high-performance introspection mechanism neccessary to achieve these higher security implementations.

Any potential NetIM implementation must overcome one specific limitation of the current KVM-based efficient introspection prototype: increasing guest overhead with snapshot frequency. Snapshotting every few seconds can be done cheaply for most loads, but snapshotting at every network packet event would not be done in the context of efficient introspection. While efficient introspection provides coherent, high-performance access to guest state, it does not support snapshotting at network packet event scale. This same type of limitation is observed in standard OS-level packet capturing and the solution is to cache network packets in a ring buffer until they can be processed. This solution is also ideal for hypervisor-based introspection because it trades the costly snapshot time for the abundant memory bandwidth available with efficient introspection.

Figure 7.5 contains a block diagram describing the process of the NetIM introspecting the guest to obtain a list of packets passed by the outgoing guest firewall to compare with the list of packets observed leaving the guest by the host-based packet capture software. This implementation of NetIM with efficient introspection can be compared with the block diagram of the VProbes implementation of the NetIM from Figure 7.4. Notable differences between the two are the addition of snapshotting replacing the VProbes memory introspection interface and the instrumentation of the guest firewall with the PCAP buffer that facilitates trading time for memory bandwidth.

Limitations of NetIM

Installing a ring buffer inside of the guest firewall inside of the guest means that the introspection is only measuring which packets were passed by the guest firewall, not the integrity of the guest firewall. A rootkit in the guest could subvert the behavior of the guest firewall to pass packets created by the rookit and they would appear normal to the NetIM. The NetIM only exposes packets that hid by inserting themselves behind the guest firewall, it makes no guarantees about the provenance of packets that do not appear hidden. Other guest- or introspection-based security mitigations will have to step in and fill this possible gap.

7.3.3 Performance Evaluation of NetIM with Efficient Introspection

Evaluating the potential performance of the NetIM with efficient introspection must begin with understanding the requirements for supporting guest packet capture with the ring buffer. Then the specific limitations on performance imposed by the NetIM must be examined in the context of our knowledge of efficient introspection behavior informed by the microbenchmarking analysis.

In order for the NetIM to recover guest network packet information, the information must be cached in the ring buffer by the guest firewall. The potential NetIM application only retrieves network packet information during a snapshot event. The ring buffer must be sized appropriately for all the packet information to be successfully stored between hypervisor snapshots.

The exact sizing of the buffer can be estimated by multiplying the time between snapshot events by the network traffic rate of the guest. Figure 7.6 contains a chart illustrating the memory requirements for buffering outgoing network packets at various performance levels and snapshotting periods. Three performance level are plotted: slow, 1 MB/s, which is actually faster than required for most desktop computing tasks like online banking or writing documents; medium, 10 MB/s, which represents the requirements of streaming video online; and, fast, 700 MB/s, which was the fastest network transfer rate observed from our test platform. The chart was limited to only display

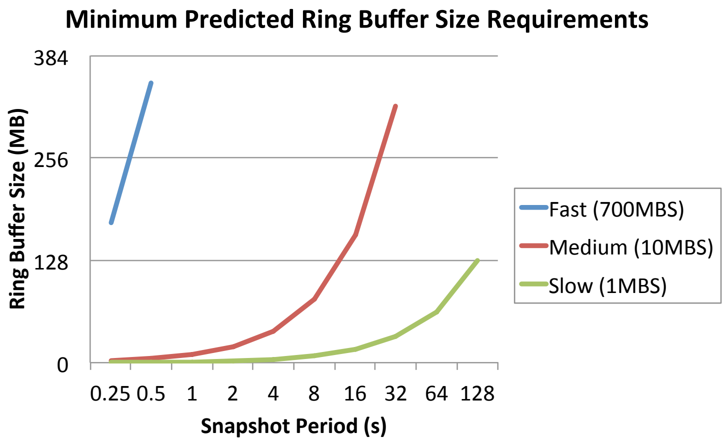


Figure 7.6: Chart illustrating the memory requirements to buffer outgoing network packets for analysis by the NetIM.

ring buffer sizes less than 512 MB, or one-quarter of the total VM size, as spending that much memory to support the NetIM application would represent an undue burden on normal guest operation. The same general trend is seen for all three performances levels, at low periods the minimum ring buffer size requirements are small but the ring buffer size requirements as snapshots become further apart in time.

The packet capture ring buffer size imposes memory overheads on the guest, but also effects guest performance in other ways. Ring buffer size limits the maximum time between snapshots. For the slow network performance level shown in Figure 7.6 this maximum time is approximately a minute between snapshots for a 64 MB buffer, but for the fast performance level the smallest time between snapshots is 0.5 seconds for a reasonably sized buffer. The number of bytes written into the ring buffer will also affect snapshot stop time performance. Like any other dirty pages, the pages written into the ring buffer will have to be copied into the snapshot memory during the snapshot by the Delta-Copy snapshot mechanism.

These application specific limitations – ring buffer memory overhead, ring buffer capacity, and ring buffer effect on snapshot performance – must be examined in the context of snapshot performance in order to determine the feasability of the potential NetIM application. Figure 7.7 contains the efficient introspection Performance Estimate heat map overlayed with NetIM-specific restrictions. As with the antivirus scanner, the time available between snapshots for the introspection application to actually measure the packet capture buffer may be insufficient for short snapshot

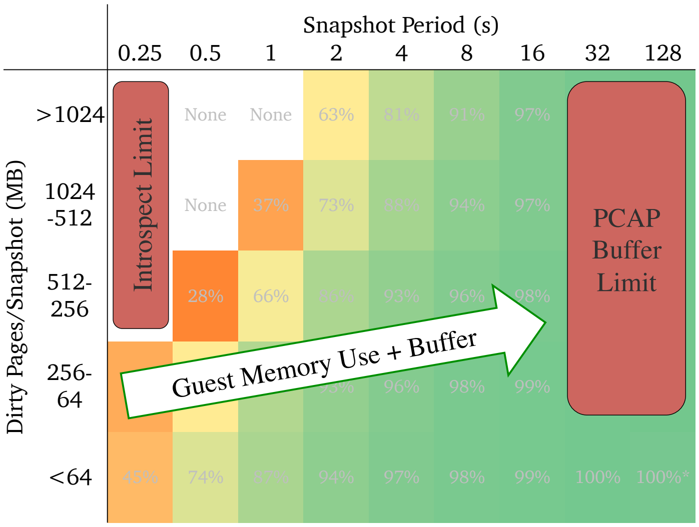


Figure 7.7: efficient introspection performance heat map overlayed with NetIM specific limitations.

periods. At the longer end of the snapshot period scale, the memory overhead of capturing large amounts of packets in the ring buffer becomes infeasible. Where the introspection and buffer capacities are feasible, it must be noted that the guest memory dirty page load must account for both the guest load but also the packet capture buffer.

Limitations of NetIM

The Network Integrity Manger provides protections against software that subverts an OS-based firewall by inserting itself below the firewall, it provides no protection against software that disables the firewall directly. The addition of the ring-buffer to the guest firewall also means that the guest must be modified for high performance with the NetIM security application. The intention of this proposed work is to strengthen the assumptions made by OS-level security protection mechanisms, not to replace OS-level security.

7.4 Application Summary

Two potential applications, signature-based Antivirus and NetIM, could not be implemented using introspection support tools available before efficient introspection. Both applications were were evaluated to determine whether the performance of the efficient introspection is sufficient to support interesting security applications. The performance of each security application was estimated by comparing the memory writing and snapshot frequency requirements of the applications against the performance of microbenchmark evaluations under conditions with similiar memory writing and snapshotting frequencies. The evaluation of these potential applications serves as guidance for potential developers interested in developing new security applications that take advantage of efficient introspection.

Snapshot stop time is a significant source of performance overhead in efficient introspection. In the case of the NetIM, the guest firewall had to be modified to store outgoing packets in a ring buffer because snapshotting frequencies could not be scaled to meet the application requirements. These modifications enabled the NetIM application to trade snapshot stop time costs for memory requirements. Trading memory for time exploits the guest-memory introspection performance of efficient introspection while minimizing the performance impact of snapshot stop time and should be applied to all future applications implemented using efficient introspection.

The Antivirus and NetIM case studies presented in this chapter suggest that the performance of the efficient introspection should be sufficicent to carry out useful security applications while maintaining normal guest operation for a variety of guest loads.

Chapter 8

Related Work

Hypervisor Introspection

The primary application of efficient introspection is to increase the performance of dynamic ad-hoc introspection techniques [6, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30] and examining existing systems will provide insights into possible pitfalls in their implementation and evaluation. Traditional antivirus software relies on signature checking and signature checking techniques have been applied from the high-ground position of a virtualized environment [31, 16, 32, 33, 18, 34]. efficient introspection will increase the performance of memory scanning and, thereby, the speed at which memory signatures can be calculated and evaluated for malware. efficient introspection will benefit semantic malware detection techniques like virtualization-based system call analysis [35, 36, 37] by providing fast, coherent access to system call arguments and system state at the time of call-gate execution. Kernel state validation from the hypervisor [38, 39, 40, 10] typically validates only a small subset of the kernel state to maintain real-time performance but efficient introspection will enable increased validation.

Real-time instruction-grain monitoring a logging techniques offered by tools like Valgrind [41] supports fine-grained instruction and memory flow checks like memory taintedness tracking and checking for references to uninitialized memory through small atomic checks. These appraoches tightly couple monitoring and checking to instruction execution in the monitored CPU and incur substantial overhead in software-only implementations. Hardware support for real-time instruction-grain monitoring such as Log Base Architectures (LBA) [42] and Dynamic Instruction Stream Editing (DISE) [43] promise normal process performance by augmenting the processor with hardware-assisted propogation tracking support. In contrast to these approaches, efficient introspection supports large checks by decoupling monitoring from execution, thereby reducing developer overhead and enabling new classes of checking that are too inefficient even with the hardware assisted tracking supports. These two approaches – instruction grain tracking versus system-level snapshotting – are complimentary, providing different answers to different problems but both increase overall security while maintaining performance.

Hypervisor Replay

Examining fast hypervisor-based replay [44, 45, 46] techniques will provide important insight into the development of high performance efficient introspection implementations. One critical difference between replay and efficient introspection operation is that efficient introspection operates in realtime and does not require saving complete system state for future replay. The efficient introspection relies on quickly generating VM-guest memory snapshots and newly proposed architectural features like RowClone by Seshadri et al. [47] could benefit efficient introspection by efficiently copying memory pages directly in-DRAM, significantly reducing memory bandwidth and CPU overhead penalties.

Hidden Process Detection

The Lycosid hidden process detection mechanism developed by Jones et al. [48] provided the inspiration for the Network and Disk Integrity Managers. Lycosid detects hidden processes by comparing the guest reported process list (*top*) with a list generated by the hypervisor observing guest address space creations and deletions associated with process creation and destruction. One significant difference between the Integrity Managers and Lycosid is that the Integrity Managers can directly observe every single network or disk access while Lycosid must measure the processes running on the guest indirectly. This allows the Integrity Manager to directly compare the reported and observed lists but Lycosid must use statistical comparison techniques to detect the existence of hidden processes.

Semantic Gap

The system state context lost between studying the operating system from within itself versus examining an operating system from the privileged hypervisor is frequently referred to as a semantic gap. The semantic gap has been studied in depth [49, 50, 51, 40], revealing the neccessity for higher performance tools that allow deeper introspection with acceptable performance that can bridge that semantic gap. The Volatility Framework [52] is a set of python tools for extracting context from volatile RAM samples. The Volatility Framework has forensic capabilities for analyzing RAM samples from Windows, Linux, Android, and MacOS and extracting details like process listing, listing network interfaces, and listing mounted devices. The LibVMI library can act as an interface between The Volatility Framework and running virtual machine guests.

Rootkit Detection

Existing kernel-resident behavioral Windows API instrumentation, such as TTAnalyze by Bayer et. al. [53], CWSandbox [54], Anubis by Bayer et. al. [55], and Process Implanting by Zhongshu et. al. [56], provide guidance for the design of Windows API instrumentation implemented at the hypervisor level.

Virtualization supported behavioral malware detection software, such as Ether by Dinaburg et. al. [26], demonstrate the capability of non-resident guest monitoring for monitoring guest behavior and the Integrity Manager expands the scope of out-of-guest monitoring.

VMDetector by Wang et. al. [57] uses multi-view rootkit technology but relies on in-guest instrumentation that can be corrupted by the virus under analysis. Patagonix by Litty et. al. [58] acts in a similiar manner to VMDetector but adds program signature matching.

Hypervisor-Guest Isolation

Managed virtual appliances like MokaFive Live PC [59], VMware Player [60], Invincia [61], Oracle Virtual Box [62], and Citrix Xen App [63] provide users with familiar computing environments for running their applications while freeing them from performing system management tasks like software updates. The Qubes OS by Rutkowska and Marczykowski also supports virtual appliances but focuses on security sandboxing and disposable guests [64]. The efficient introspection prototype will build on an existing hypervisor to provide additional security protection against rootkits and viruses. It is the authors' hope that, in the future, Virtual Appliance developers will be able to increase the security of their customers computing by implementing efficient introspection-enhanced security modules appropriate to their specific applications. Other work developing a trusted computing base [65, 66], implementing instruction trace construction [67, 68], dynamic taint tracking [27, 69, 70, 71] or exploiting the isolation properties of virtualization [72] may not benefit directly from efficient introspection but systems designed to exploit these properties could apply efficient introspection accelerated security applications to increase security as needed using the virtualization platform already in place.

InZero [73] is a commercial security product which provides an entirely seperate computer on which to perform critical computing tasks that is securely accessed from the consumers usual computer. The physical seperation provided by InZero's product increases isolation but at significant cost in duplication in hardware. Virtual Appliances provide less isolation than duplicated machines, but protections, like those propsed by the Integrity Manager, provide increased guarantees against persistent, hidden infections.

Chapter 9

Conclusions

This thesis creates a new efficient introspection platform that enables the development of new classes of introspection applications that were previously rejected as slow or expensive to develop.

9.1 The Need for Efficient Snapshotting

Rootkits are a class of malicious software that operate with operating system-level privilege and evade detection by subverting operating system-level mitigations. Hypervisor-based introspection operates at a higher privilege level than the guest operating system and provides a high-ground position for malicious software protection isolated from potential malicious software in the guest. This thesis develops three requirement for efficient introspection:

1. coherent memory views between the host and guest,
2. native memory introspection performance,
3. normal guest performance.

Coherent memory views are required to access the complete state of the guest at a single moment in time and simplify the development of introspection applications. Native memory introspection performance is required to support introspection applications that perform extensive inspection of the entire guest memory, like full memory antivirus signature scanning, within reasonable periods of time. Normal guest performance ensures that efficient introspection applications will not be rejected for placing an undue burden on the end-users of the introspected system. The efficient introspection platform developed in this thesis combines all three of these properties by decoupling guest operation from introspection through high-performance memory snapshotting.

9.2 Summary of Work

Requirements for Efficient Introspection: This thesis develops three requirements for efficient introspection: coherent memory views between the host and guest, native memory introspection performance, and normal guest performance. These requirements are necessary to provide support for a broad array of introspection applications without imposing heavy burdens upon security end-users. These requirements provide broad guidance across specific implementations of hypervisor introspection.

Fast Snapshotting: This thesis presents the case that high-performance memory snapshotting provides a practical solution to satisfying the three requirements for efficient introspection. Three methods for implementing fast snapshotting are presented and evaluated: Stop-Copy snapshotting, Delta-Copy snapshotting, and Pre-Copy snapshotting.

Application Benchmarking Evaluation: This thesis demonstrates that a prototype implementation of efficient introspection provides guest performance for a selection of application benchmarks. These applications include: Kernel compilation, ClamAV Antivirus scan, Apache Web Server, Netperf network performance, and Weka machine learning. The performance of these application benchmarks under introspection demonstrates that normal guest operation can be attained for a variety of applications.

Microbenchmarking Evaluation: This thesis presents a microbenchmark evaluation of the efficient introspection prototype. The microbenchmark evaluation allows for a systematic evaluation of various snapshotting, introspection, and guest load behavior parameters. Further, the microbenchmark evaluation provides explanation of why the application benchmarks performed well.

Potential Applications: This thesis presents two potential applications of efficient introspection: a memory-signature-based antivirus scanner and hidden network packet scanner. These two potential applications exploit the high-ground-position of hypervisor based introspection. Potential performance is analyzed by comparing the specific properties of each application with the results observed in the systematic microbenchmark evaluation. Specific performance advantages of implementing the potential application with efficient introspection instead of previously available platforms were shown.

9.3 Concluding Remarks

Celebrity photo leaks [74], financial data breaches at major retailers [75], and cyberwarefare [76] have placed computer security and privacy concerns squarely into popular culture. Traditional mitigations like signature-based antivirus software and firewalls have been widely deployed and yet these threats persist. Computer science and engineering will have to meet these concerns by providing new security techniques to help control these threats and increase public trust in computing.

Hypervisor-based antivirus is one of these techniques. The hypervisor combines total control over guest operation with isolation from the guest execution to create a high-ground position from which to provide security protections with introspection. Efficient introspection holds normal guest operation as a first-class requirement and therefore provides the benefits of hypervisor-based antivirus without placing undue burdens on the performance of the user applications

Adoption of the cloud, desktop virutalization, and even virtualization layers in mobile and gaming consoles, are increasing the number of installed virtualization platforms. New platforms and applications of virtualization increase the relevancy of introspection and exands creates new opportunities for hypervisor-based security protection. It is my hope that the efficient introspection capabilities at the heart this thesis will be carried beyond these limited demonstrations to enable new applications that will create a secure computing future.

9.4 Future Work

While Delta-Copy snapshotting can reduce snapshot-memory copy time very significantly, non-copy snapshotting costs related to stopping and restarting the guest remain a barrier to efficient snapshotting. The reasons for the slow-snapshot pause-restart time demonstrated in this thesis are specific to the implementation of the KVM hypervisor. KVM is a split virtualization platform where most events are handled directly by the bare-metal processor, but certain events are passed off to an emulator. For efficiency, the emulator caches certain guest-state, like some memory, locally to the emulator. One of the major events taking place during a VM pause event is synchronizing the locally held emulator state back into the global guest-state data structures. Flushing these caches back into global state is critical to satisfying the introspection requirement of coherent memory access. Future research into revealing the underlying causes of these non-copy snapshotting bottlenecks and resolving them could enable finer-grained snapshotting than is currently possible.

This thesis presents an implementation of efficient snapshotting as an extension of the KVM hypervisor. The properties of KVM that are exploited in the extension of are all required to provide efficient guest migration: pause-resume capability, dirty-page tracking, and memory introspection. Most commonly available hypervisors offer these properties and these hypervisors could be extended to support introspection. Future work could measure the efficiency of other hypervisors in implementing these properties when contemplating adaptation of efficient snapshotting to other platforms.

In the current implementation, snapshots cannot be synchronized to events taking place within the guest or hypervisor. Some possible events are: guest accesses to specific memory addresses, guest execution of specific memory addresses, guest system events like page table misses, guest disk accesses, guest network accesses, and hypervisor handled page table misses. Further research might reveal even more interesting events. The introspection application would configure the hypervisor to snapshot at the next trigger event and then call-back to the introspection application. Implementing event-triggered snapshotting capability would enable new classes of introspection applications introspect. Event-triggers might even create opportunities for extension of the application domain of snapshotted introspection beyond security to debugging.

Initial Open-Source release to the LibVMI project already began with Stop-Copy snapshotting but Delta-Copy snapshotting is planned. The LibVMI project already includes a basic implementation of the Stop-Copy snapshotting mechanism. The release is packaged as an Open-Source patch to the KVM hypervisor. Delta-Copy snapshotting KVM-patch may be released to the LibVMI project. Currently the patch to the KVM hypervisor is applied and compiled by LibVMI end-users. The patch could also be released beyond the LibVMI project as a contribution directly to the KVM Project. Exploration into the feasibility and acceptance of this type of extension of the KVM Project is left for future work.

In this thesis snapshot-stop time and snapshot period are identified as key factors effect normal guest performance. Benchmark evaluations evaluated guest performance in the context of fixed snapshot periods. These snapshot periods do not have to be fixed and could be dynamically adjusted to meet guest service requirements. The snapshot stop-time could be predicted by measuring the number of dirty pages that would need to be copied, and the snapshot could be delayed by a corresponding factor to ameliorate stop time. Future work could develop this type of service-level-guaranteeing introspection.

Instrumenting guests by allowing introspection applications to modify guest state is not currently supported by the shared efficient snapshot mechanism. Maintaining coherency between the guest state and introspection application will become more difficult as currently the introspection application is provided read-only access to the guest. Future research could examine mechanisms that support dynamic, runtime guest instrumentation while still satisfying the requirements for efficient introspection.

Full VM snapshotting may not be necessary for all applications. Significant performance gains could be realized if only certain portions of guest memory state had to be snapshotted. Partial snapshotting would be espicially useful for guest loads which write to significant proportions of their memory space, where the Delta-Copy snapshotting mechanism does not show significant improvements over Stop-Copy. Future work into the feasibility of partial snapshots will have to begin with an exploration of appropriate introspection applications, like statistical sampling methods, that will not have their security properties compromised by only accessing partial guest state.

Snapshotting guest memory for introspection may map favorably onto the well tread field of memory transactions. The decoupled introspection thread may begin it's introspection process by opening a memory transaction for the guest memory region on behalf of the guest. The shared memory will then appear snapshotted to the introspection thread until the introspection completes and the guest memory transaction completes, committing the transactions into the updated snapshot. Software implementations of memory transactions like RVM by Satyanarayanan et al. [77] show that memory transactions of this type can be efficient on timescales shorter than the snapshot periods explored in this work without incurring prohibitive code complexity. Hardware-accelerated implementations like Transactional Memory Coherency and Consistency by Hammond et al. [78] should be examined in the context of the newly released Intel Transactional Synchronization Extensions [79]. Future work into applying the lessons from memory transactions to creating high-performance memory snapshots may provide a way towards lessening the memory cost of efficient introspection while maintaining coherence and normal performance.

Rootkits and other malicious software have developed countermeasures to security detection that include self-modification and self-encryption that present a smaller footprint in memory for detection. Malicious software has been observed to detect the presence an imminent and further reduce it's footprint. Future work into examining whether snapshots can be detected by an in-guest rootkit and whether that snapshots timing can be hidden from the guest to prevent possible malware from taking evasive actions.

Guests frequently contain duplicated pages. The current implementation of snapshotting makes no attempt to identify these duplicates. In fact, to ensure coherency, the snapshotting mechanism copies each guest page individually using mechanisms that may prevent OS- or hypervisor-level de-duplication mechanisms from functioning. Future work will examine whether de-duplication could reduce the overhead of the snapshot memory which is currently 100% of the snapshotted guest.

Dirty page tracking could inform introspection application scanning behavior. For example, the signature-based antivirus scanner could consult the dirty page list for a new snapshot and only re-scan pages that had been changed since their previous scan. Future work could explore introspection applications that could benefit from this capability and efficient ways to communicate the dirty page list from the hypervisor to the introspection applicaiton.

Currently the Pre-Copy mechanism relies on a thread that periodically updates the dirty-page list managed by the hypervisor kernel module to select dirty pages for pre-copying, clears them from the list, and copies them. This method is inefficient as the list of pages is slow to synchronize, adversely affects guest performance, and must be iterated page-by-page to identify dirty pages. An alternative implementation of the pre-copy which relied on the hypervisor dirty page call-back mechanism to identify dirty pages directly might be more efficient and could be explored in future work.

Bibliography

[1] B. D. Payne, “VMITools - Virtual Machine Introspection Tools.” http://code.google.com/p/vmitools/, 2013.

[2] T. K. Project, “Kernel samepage merging,” 2014.

[3] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “kvm: the Linux virtual machine monitor,” in *Ottawa Linux Symposium*, pp. 225–230, July 2007.

[4] C. A. Waldspurger, “Memory resource management in vmware esx server,” *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 181–194, Dec. 2002.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP ’03, (New York, NY, USA), pp. 164–177, ACM, 2003.

[6] B. Payne, M. de Carbone, and W. Lee, “Secure and flexible monitoring of virtual machines,” in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pp. 385 –397, dec. 2007.

[7] G. Team, “Stealth MBR rootkit.” http://www2.gmer.net/mbr/, Jan. 2008.

[8] H. Lau, “Are mbr infections back in fashion? (infographic).” http://www.symantec.com/connect/blogs/are-mbr-infections-back-fashion-infographic, Aug. 2011.

[9] P. Kleissner, “Analysis of mebratrix.” http://web17.webbpro.de/index.php?page=analysis-of-mebratix, 2010.

[10] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, (New York, NY, USA), pp. 555–565, ACM, 2009.

[11] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Vmm-based hidden process detection and identification using lycosid," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, (New York, NY, USA), pp. 91–100, ACM, 2008.

[12] P. Ször and P. Ferrie, "Whitepaper: Hunting For Metamorphic." http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf, 2003.

[13] C. Staelin and H. packard Laboratories, "lmbench: Portable tools for performance analysis," in *In USENIX Annual Technical Conference*, pp. 279–294, 1996.

[14] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," *SIGPLAN Not.*, vol. 47, pp. 279–290, Mar. 2011.

[15] Q. Team, "Qemu." http://qemu.org, 2013.

[16] 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, (Berkeley, CA, USA), pp. 13–13, USENIX Association, 2004.

[17] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, (New York, NY, USA), pp. 335–350, ACM, 2007.

[18] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, (New York, NY, USA), pp. 566–577, ACM, 2009.

[19] R. Jones, "The netperf homepage." http://www.netperf.org/, 2012.

[20] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 2008.

[21] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, (New York, NY, USA), pp. 477–487, ACM, 2009.

[22] F. Baiardi and D. Sgandurra, "Building trustworthy intrusion detection through vm introspection," in *Proceedings of the Third International Symposium on Information Assurance and Security*, IAS '07, (Washington, DC, USA), pp. 209–214, IEEE Computer Society, 2007.

[23] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," *SIGPLAN Not.*, vol. 47, pp. 133–144, Mar. 2012.

[24] N. A. Quynh and K. Suzaki, "Xenprobes, a lightweight user-space probing framework for xen virtual machine," in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, (Berkeley, CA, USA), pp. 2:1–2:14, USENIX Association, 2007.

[25] 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*, VMSec '09, (New York, NY, USA), pp. 1–10, ACM, 2009.

[26] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, (New York, NY, USA), pp. 51–62, ACM, 2008.

[27] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, (New York, NY, USA), pp. 65–74, ACM, 2007.

[28] B. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 233 –247, May 2008.

[29] P. M. Chen and B. D. Noble, “When virtual is better than real,” in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS ’01, (Washington, DC, USA), pp. 133–, IEEE Computer Society, 2001.

[30] D. G. Murray, G. Milos, and S. Hand, “Improving xen security through disaggregation,” in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE ’08, (New York, NY, USA), pp. 151–160, ACM, 2008.

[31] X. Jiang, X. Wang, and D. Xu, “Stealthy malware detection through vmm-based ”out-of-the-box” semantic view reconstruction,” in *Proceedings of the 14th ACM conference on Computer and communications security*, CCS ’07, (New York, NY, USA), pp. 128–138, ACM, 2007.

[32] M. Payer and T. R. Gross, “Fine-grained user-space security through virtualization,” *SIGPLAN Not.*, vol. 46, pp. 157–168, Mar. 2011.

[33] S. Ghosh, J. Hiser, and J. W. Davidson, “Replacement attacks against vm-protected applications,” in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE ’12, (New York, NY, USA), pp. 203–214, ACM, 2012.

[34] N. R. Paul, *Disk-level behavioral malware detection*. PhD thesis, University of Virginia, Charlottesville, VA, USA, May 2008. AAI3312124.

[35] N. A. Quynh and Y. Takefuji, “Towards a tamper-resistant kernel rootkit detector,” in *Proceedings of the 2007 ACM symposium on Applied computing*, SAC ’07, (New York, NY, USA), pp. 276–283, ACM, 2007.

[36] M. Laureano, C. Maziero, and E. Jamhour, “Intrusion detection in virtual machine environments,” in *Proceedings of the 30th EUROMICRO Conference*, EUROMICRO ’04, (Washington, DC, USA), pp. 520–525, IEEE Computer Society, 2004.

[37] T. Garfinkel, “Traps and pitfalls: Practical problems in system call interposition based security tools,” in *In Proc. Network and Distributed Systems Security Symposium*, pp. 163–176, 2003.

[38] R. Riley, X. Jiang, and D. Xu, “Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing,” in *Proceedings of the 11th international symposium on Recent*

Advances in Intrusion Detection, RAID ’08, (Berlin, Heidelberg), pp. 1–20, Springer-Verlag, 2008.

[39] M. Neugschwandtner, C. Platzer, P. M. Comparetti, and U. Bayer, “danubis: dynamic device driver analysis based on virtual machine introspection,” in *Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment*, DIMVA’10, (Berlin, Heidelberg), pp. 41–60, Springer-Verlag, 2010.

[40] Y. Fu and Z. Lin, “Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection,” in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 586 –600, May 2012.

[41] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’07, (New York, NY, USA), pp. 89–100, ACM, 2007.

[42] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, “Flexible hardware acceleration for instruction-grain program monitoring,” in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA ’08, (Washington, DC, USA), pp. 377–388, IEEE Computer Society, 2008.

[43] M. L. Corliss, E. C. Lewis, and A. Roth, “Dise: A programmable macro engine for customizing applications,” *SIGARCH Comput. Archit. News*, vol. 31, pp. 362–373, May 2003.

[44] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, “Flashback: a lightweight extension for rollback and deterministic replay for software debugging,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC ’04, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2004.

[45] O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh, “Dejaview: a personal virtual computer recorder,” *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 279–292, Oct. 2007.

[46] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen, “Multi-stage replay with crosscut,” *SIGPLAN Not.*, vol. 45, pp. 13–24, Mar. 2010.

[47] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. M. u, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data,” Tech. Rep. CMU-CS-13-108, Computer Science Department, School of Computer Science, Carnegie Mellon University, April 2013.

[48] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “VMM-based hidden process detection and identification using Lycosid,” in *VEE ’08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, (New York, NY, USA), pp. 91–100, ACM, 2008.

[49] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, “Virtuoso: Narrowing the semantic gap in virtual machine introspection,” in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP ’11, (Washington, DC, USA), pp. 297–312, IEEE Computer Society, 2011.

[50] B. Dolan-Gavitt, B. Payne, and W. Lee, “Tech report gt-cs-11-05: Leveraging forensic tools for virtual machine introspection.” http://hdl.handle.net/1853/38424, Nov. 2011.

[51] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, “Dksm: Subverting virtual machine introspection for fun and profit,” in *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS ’10, (Washington, DC, USA), pp. 82–91, IEEE Computer Society, 2010.

[52] M. Auty, A. Case, M. Cohen, B. Dolan-Gavitt, M. Hale Ligh, J. Levy, and A. Walters, “The volatility framework v2.3.” http://code.google.com/p/volatility/, 2013.

[53] U. Bayer, C. Kruegel, and E. Kirda, “Ttanalyze: A tool for analyzing malware,” Apr. 2006.

[54] C. Willems, T. Holz, and F. Freiling, “Toward automated dynamic malware analysis using cwsandbox,” *Security Privacy, IEEE*, vol. 5, pp. 32 –39, march-april 2007.

[55] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "A view on current malware behaviors," in *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, LEET'09, (Berkeley, CA, USA), pp. 8–8, USENIX Association, 2009.

[56] 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*, pp. 147 –156, oct. 2011.

[57] Y. Wang, C. Hu, and B. Li, "Vmdetector: A vmm-based platform to detect hidden process by multi-view comparison," in *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, pp. 307 –312, nov. 2011.

[58] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proceedings of the 17th Conference on Security Symposium*, SS'08, (Berkeley, CA, USA), pp. 243–258, USENIX Association, 2008.

[59] moka5, Inc., "MokaFive Suite Overview." http://www.moka5.com, Jan. 2011.

[60] VMware, Inc., "VMware Virtual Appliance Marketplace: Virtual Applications for the Cloud." http://www.vmware.com/appliances, Jan. 2011.

[61] Invincea, Inc., "White paper: Web malware explosion requires new protection paradigm." http://www.invincea.com/images/uploads/INV_Malware_WP_FW.pdf, Mar. 2010.

[62] Oracle, Inc., "VirtualBox." http://www.virtualbox.org, Jan. 2011.

[63] Citrix Systems, Inc., "Citrix XenApp - Product Overview." http://www.citrix.com/site/resources/dynamic/salesdocs/XenApp6/Product_Overview.pdf, Jan. 2011.

[64] J. Rutkowska and M. Marczykowski, "Welcome to the Qubes OS Project." http://qubes-os.org, Mar. 2013.

[65] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, “Bitvisor: a thin hypervisor for enforcing i/o device security,” in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE ’09, (New York, NY, USA), pp. 121–130, ACM, 2009.

[66] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, “Terra: a virtual machine-based platform for trusted computing,” *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 193–206, Oct. 2003.

[67] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, “V2e: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis,” *SIGPLAN Not.*, vol. 47, pp. 227–238, Mar. 2012.

[68] X. Hu, T.-c. Chiueh, and K. G. Shin, “Large-scale malware indexing using function-call graphs,” in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS ’09, (New York, NY, USA), pp. 611–620, ACM, 2009.

[69] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, “Dynamic spyware analysis,” in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC’07, (Berkeley, CA, USA), pp. 18:1–18:14, USENIX Association, 2007.

[70] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: capturing system-wide information flow for malware detection and analysis,” in *Proceedings of the 14th ACM conference on Computer and communications security*, CCS ’07, (New York, NY, USA), pp. 116–127, ACM, 2007.

[71] H. Yin and D. Song, “Technical report: Temu: Binary code analysis via whole-system layered annotative execution.” `http://techreports.lib.berkeley.edu/accessPages/EECS-2010-3.html`, Jan. 2011.

[72] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, “Cells: a virtual mobile smartphone architecture,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP ’11, (New York, NY, USA), pp. 173–187, ACM, 2011.

[73] Philip Zimmermann, “A brief assessment of the InZero security gateway.” http://www.inzerosystems.com/wp-content/uploads/2010/05/A-brief-assessment-of-the-InZero-security-gateway-Philip-Zimmermann.pdf, Dec. 2009.

[74] C. Boren, “Olympic gymnast mckayla maroney says leaked racy photos are fake, fends off twitter backlash.” http://www.washingtonpost.com/blogs/early-lead/wp/2014/09/01/olympic-gymnast-mckayla-maroney-says-leaked-racy-photos-are-fake-fends-off-twitter-backlash/, Sept. 2014.

[75] B. Krebs, “In home depot breach, investigation focuses on self-checkout lanes.” http://krebsonsecurity.com/2014/09/in-home-depot-breach-investigation-focuses-on-self-checkout-lanes/, Sept. 2014.

[76] S. Weinberger, “Computer security: Is this the start of cyberwarfare?,” *Nature*, vol. 474, pp. 142–145, June 2011.

[77] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler, “Lightweight recoverable virtual memory,” *ACM Trans. Comput. Syst.*, vol. 12, pp. 33–57, Feb. 1994.

[78] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, “Transactional memory coherence and consistency,” *SIGARCH Comput. Archit. News*, vol. 32, pp. 102–, Mar. 2004.

[79] Intel Corporation, “Intel 64 and IA-32 Architectures Optimization Reference Manual. Chapter 12: INTEL TSX RECOMMENDATIONS,” 2014.