PERFORMANCE IMPLICATIONS OF SYSTEM MANAGEMENT MODE

Brian Delgado†, Karen L. Karavanic
Portland State University
bdelgado, karavan@cs.pdx.edu

ABSTRACT

System Management Mode (SMM) is a special x86 processor mode that privileged software such as kernels or hypervisors cannot access or interrupt. Previously, it has been assumed that time spent in SMM would be relatively small and therefore its side effects on privileged software were unimportant; recently, researchers have proposed uses, such as security-related checks, that would greatly increase the amount of runtime spent in this mode. We present the results of a detailed performance study to characterize the performance impacts of SMM, using measurement infrastructure we have developed. Our study includes impact to application, system, and hypervisor. We show there can be clear negative effects from prolonged preemptions. However, if SMM duration is kept within certain ranges, perturbation caused by SMIs may be kept to a minimum.

I. INTRODUCTION

System Management Mode (SMM) is a special x86 processor mode that privileged software such as operating systems or hypervisors cannot access. This hardware feature was originally developed for operating system-independent functionality such as power throttling, hardware emulation, and running OEM code. The key distinction of SMM is its invisibility to the kernel and the hypervisor. The entry to SMM is through a System Management Interrupt (SMI), a unique type of interrupt that is much more disruptive than a traditional interrupt: When an SMI occurs, the standard behavior is all of the processor cores will enter System Management Mode. [21][7] The SMI handler will then perform the requested work, restore the interrupted context, and return. Because all CPU threads stay in SMM until the completion of the SMI's work, the severity of the impact increases with the number of cores.

In the past it has been assumed that time spent in SMM would be relatively small and therefore its side effects were unimportant. Guidelines existed only in the form of informal rules of thumb limiting the total amount of time that should be taken by each SMI. Three recent trends have increased the importance of studying the performance impacts of SMM: security integrity checker codes called Runtime Integrity Measurement Mechanisms (RIMMs), virtualization, and the trend toward an increasing number of cores per socket. In the security realm, proposals to repurpose SMM for quick detection of malware and rootkits to limit their damage [3][25][29] as well as providing secure isolated execution environments [4] dramatically change expectations over its use. Since SMM completely pauses host software execution for the duration of its work and the time required for these new usages exceeds common SMI durations, there are clear performance concerns. In recent years, applications have moved from running on native operating systems where they were impacted by other processes as well as the operating system to running within virtualized environments which added virtualization-level impacts. SMM RIMMs would cause another source of impact on applications as well as the virtualized environments on which they run. Applications running under hypervisors watched by an SMM RIMM would experience the combined impacts of each layer.

There is currently very little available data on the performance effects of SMM. Our work addresses this gap by providing a measurement methodology that enables performance analysis of SMIs, including identification of effects at both the system and application levels due to prolonged preemptions of the system; and quantification of the resulting performance impacts at varying levels of system preemption. We present results demonstrating significant impacts at both the system and application levels.

II. BACKGROUND

A. System Management Mode

Intel introduced SMM with the Intel 386 SL processor and it provides "an alternate operating environment that can be used to monitor and manage various system resources for more efficient energy usage, to control system hardware, and/or to run proprietary code." [14] AMD x86 CPUs also feature SMM [2] and Intel's Itanium CPU features a PMI that is similar in concept to an SMI. [15] SMM is designed such that neither privileged software nor applications can inspect its memory (SMRAM) [14] or directly detect time spent in this mode. SMIs can occur for a variety of reasons including: reporting of hardware errors, thermal throttling, power capping, and system health checks. [16] SMIs can be synchronous via a CPU instruction or asynchronous from the chipset. [8] The potential exists for an SMI to preempt time-sensitive code (e.g. code holding a global lock on one node in a cluster), resulting in delays well beyond what the software developer may have expected.

The x86 architecture features a variety of different types of exceptions and interrupts. SMIs are unique in that they are a higher priority interrupt than Nonmaskable Interrupts (NMIs) and device interrupts. [14] SMM has the

† *The author was a full-time employee of Intel Corp. while this work was done.*

978-1-4799-055-3/13/$31.00 ©2013 IEEE

benefit that other interrupts will not preempt it, but has the side effect that other device interrupts will only be handled after it has finished its work. [3] Intel has released a tool called the "Intel BIOS Implementation Test Suite" (BITS) [22] that counts and measures SMIs occurring on a system and checks that their latencies are within "acceptable" limits (currently defined as 150 microseconds). This rule of thumb has been the only available guideline for latency tolerance.

B. Implementing Runtime Integrity Measurements in System Management Mode

In recent years, many IT organizations and end-users have begun using virtualization; however because hypervisors operate at a low level on the platform and have visibility into all virtual machines running on the system, they are unfortunately a tempting target for malicious attacks. For example, security researchers have shown a layered attack on the Xen hypervisor designed to install a stealth backdoor. [27] The attack replaces the contents of a privileged hypervisor interface (Xen hypercall) with malicious code. The attack also alters the debug exception handler to detect and execute code contained in malicious packets. The debug exception occurs before the firewall begins filtering packets, and Domain 0 never sees the packet.

The goal of a RIMM is to provide quick notifications of attacks by scanning key portions of the hypervisor on a periodic basis and generating alerts for unexpected changes. In the example described above, a RIMM could detect this attack if it were to hash Xen hypercall code and compare the current hash of the injected code with the initial measurement or similarly detect unexpected changes in the debug exception handler. RIMMs can also enforce a security assumption that no two virtual machines should be sharing memory by examining the Xen data structure that controls domain memory allocations (the grant tables); or watch security sensitive components such as the hypervisor's code in memory, the Interrupt Delivery Table (IDT), memory segment descriptors, VM exit handlers, and Machine State Registers that can cause jumps during execution. [3]

In order to ensure that the RIMM itself isn't compromised, some have proposed that the RIMM be implemented in SMM. SMM's protected memory [27] is extremely useful for the protection of the RIMM itself. One proposed SMM RIMM, HyperSentry [3], updates the SMI handler to work with a measurement agent that runs in the hypervisor and uses SMM code to ensure that the agent has not been compromised before transferring execution to the agent. HyperSentry waits for a discrete hardware controller to generate an SMI that brings all CPU threads into SMM and directs one CPU thread to inspect the RIMM agent along with the hypervisor data structures while the other CPU threads wait. HyperCheck is another SMM RIMM that "aims to detect the in-memory, Ring-0 level (hypervisor or general OS) rootkits and rootkits in privileged domains of hypervisors." [25] HyperCheck implements a small monitoring agent in SMM to check security-sensitive values such as the CR3 register on the CPU, which controls paging. HyperCheck places its measurement agent in SMM, unlike the split model in HyperSentry. SMM times measured for each are (40 *ms*, 1 per sec) for HyperCheck and (35 *ms*, 1 per 8 secs or 1 per 16 secs) for HyperSentry.

SPECTRE is a recent SMM RIMM that examines hypervisors, operating systems, and user processes for certain attacks including heap spray, heap overflow, and rootkit detection. SPECTRE can detect heap spray attacks in 25-31 *ms*, heap overflow attacks in 32 *ms*, and the KBeast rootkit in 5 *ms*. [29] The amount of SMI latency proposed for the various SMM RIMMs is significantly longer than common rules of thumb that we described in Section A.

Besides the long and currently unbounded software preemptions and periodic interrupts, there is another concern about the SMM RIMM approach: malware could seek to evade detection by operating for short periods of time and then going dormant to lessen the chance of being observed by the SMM RIMM. [24] This implies that an SMM RIMM would need to do frequent checks as well as be scheduled to execute randomly.

C. Implementing Workload Isolation Using System Management Mode

A recent paper by Azab et al. describes a mechanism called SICE which allows for workload isolation in untrusted cloud environments. [4] SICE relies upon two key SMM features to facilitate this mechanism: SMRAM for memory protection, and SMIs as the interface to the isolated environment. The total end-to-end time required from enter to exit of the isolated environment is 67 microseconds. In SICE's multi-core mode this is a one-time overhead, but in "time-sharing" mode, the isolated execution environment is context switched with regular applications and system code on the same core, and this overhead would be incurred on every context switch.

III. Related Work

The effects of SMIs are quite unique in comparison to other interrupts occurring on the system. Traditional device interrupts can preempt running application code, however, they are able to provide acknowledgement to the hardware and set up mechanisms for future processing. [5] SMIs cannot typically be deferred in this manner and all CPU threads leave the host environment and transition into the SMI handler upon receiving an SMI. Thus they have a broader system impact than a traditional device interrupt. The operating system also isn't able to mask SMIs as it can with traditional device interrupts which results in the potential for an SMI occurring at an unexpected point in time. An Intel whitepaper noted that SMIs present serious complications to the XenoMai RTOS microkernel as they are invisible to the RTOS scheduler. [13] Latency-sensitive users attempt to disable SMIs to remove their impact [11] [30] and others use tools to detect their occurrence. [26]

The developers of both HyperSentry and Hypercheck included some performance evaluation in their proposals for SMM RIMMs, including end-to-end execution times, limited studies of the performance impact on host software, and a detailed time breakdown. Our focus by contrast is to characterize the performance impacts of the

SMM more generally and deeply, in a way that will be relevant for broader usages of SMM.

Our evaluation is similar to previous workload perturbation studies examining the effect of *noise* from software heartbeats and system daemons [19] hardware interrupts [5], and network interrupts. [23] Ferreira et al [10] have found that noise's effect on an application may be reduced by absorption; conversely, the impact of noise can be amplified when it occurs at a performance-sensitive time. Since SMIs are the highest priority interrupt, they affect the platform on a greater scale than these other types of noise. The operating system timer interrupt that Beckman et al study [5] is itself at risk from asynchronous SMI noise.

IV. Methodology

Empirically measuring the effects of SMIs involves challenges in both generating and measuring the time spent in SMM. The BIOS of commodity systems is essentially a black box to the user and its code resides in a hardware protected region, so generating the needed SMIs for this study was challenging. We developed three different SMI generation techniques for this study: Chipset-based, Blackbox SMI, and Modified BIOS.

Chipset-based. Our initial results were gathered by turning on varying frequencies of hardware-generated (chipset) SMIs using the SWSMI_TMR feature. This approach does not require access to the BIOS, however the significant shortcomings are an inability to specify precise SMI durations or generate longer SMIs than 0.11*ms* on our system. The feature is also not supported on all motherboards. We approximated longer SMIs by generating a large number of short SMIs ("short but frequent"). The caveat is that these short SMIs didn't preempt the system for the duration of a single longer SMI in one interval.

Blackbox SMI. As an improvement to *Chipset-based*, we created a device driver that called existing software SMIs by writing specific values to an IO port typically configured to generate SMIs ("APM_CNT" on Intel, "SMI Command Port" on AMD). These SMIs consume time away from host software control corresponding to the amount of work to be done which is typically longer than the durations supported in the Chipset SMI approach. In order to find longer SMIs, we created software to discover them by writing various values to APM_CNT and deriving the SMI processing time by taking timestamps before and after the SMI generation. With this method, we found several SMIs that consumed 5*ms* or more. However, our key concern with this approach is that without knowing more about what the SMI was actually doing, we couldn't rule out the possibility of performance side effects. For example, if the SMI adjusted the CPU frequency, we would be introducing a side effect into our measurements.

Modified BIOS. Greater precision requires modifying the SMI handler. This option is not typically available to end-users, but we had the ability to modify our SMI handler to allow a user-configurable amount of delay. In this approach we added twelve values that could be written to the APM_CNT port to generate varying levels of SMI delays: (in *ms*) 1.43, 5, 10, 20, 50, 99, 495, 990, 5k, 10k, 20k, 64k. When the SMI handler received control, it would delay in a loop for the specified amount of CPU cycles before returning control to the host software. In this way, all CPUs left the host software and stayed in SMM for the specified amount of time. This mechanism provided a useful way to preempt the system for a controllable duration. In order to calculate the delay length in SMM, we used the CPU's time-stamp counter (TSC). The TSC can be influenced by CPU frequency in some configurations, however many modern CPUs support "Constant TSC" to ensure that the duration of CPU clock isn't influenced by changes in CPU frequency. [14] To ensure that the delays were of the expected length we took a CPU timestamp before and after generating a long SMI, calculated the delta; and double-checked the wall clock times of the longer delays.

As we triggered our *Blackbox SMI* and *Modified BIOS* delays using an OUT CPU instruction which needs to be executed from Ring 0, we developed device drivers for each system we measured: Xen 4.1.2, Centos 6.0/6.3, and Windows Server 2012, to trigger SMI delays once a second. For Xen we used the kernel work queues to schedule our software SMI once a second. For Windows Server 2012, we used the kernel function IoStartTimer to schedule one SMI/second. Our test setup additionally allows us to generate a single SMI on demand.

V. System Level Effects

Unlike the application-level delays caused by multiprogramming, the delays caused by time in SMM represent time intervals where the processor is not under OS control. What are the effects of this "invisible" processing time? To answer this question, we investigated the effects of SMM time on the kernel, focusing on the code that immediately follows each timer interrupt.

A. Timer Interrupt Effects

Traditionally many important scheduling and statistical operations in the Linux kernel happened on a regular timer tick interval, e.g. {100, 250, 300, 1000} times a second. For power savings reasons, the "tickless kernel" option has been added, allowing the kernel to remain idle longer by avoiding unnecessary wake-ups. If the next scheduled timer event would occur after the next periodic timer tick, the kernel would reprogram "the per-CPU clock event device to this future event" allowing the CPU to remain idle longer. [20] In both traditional and tickless operation, our inspection of the Linux 3.1.4 kernel showed that once the kernel wakes, it runs several key functions in `do_timer` which update the kernel's internal clock count (jiffy) and wall clock time, and calculate the load on the system. (See Figure 1.) Then it calls `update_process_times` which charges time to executing processes, runs high resolution timers and raises SoftIRQs for local timers, checks if the system is in a quiet state for RCU callbacks, does printk statements, runs IRQ work, calls `scheduler_tick` and then runs timers that are due. [17] The `scheduler_tick` function performs several important tasks including updating scheduler timestamp data, updating timestamps for processes on the

run queue, updating CPU load statistics based on the run queue, invoking the scheduler, updating performance events for the Linux Performance Event subsystem, determining if a CPU is idle at the clock tick, and load balancing tasks between CPU run queues.

Intel technical documentation notes "All interrupts normally handled by the operating system are disabled upon entry into SMM" [14] which presents the possibility for an SMI to perturb timer interrupts and consequently impact the important scheduling operations in `scheduler_tick` as a side effect.

To investigate the degree to which SMIs preempted timer interrupts, we instrumented the Linux kernel `do_timer` and `scheduler_tick` functions. For `do_timer` we logged a trace point just after the timer interrupt occurs, recording the total number of SMIs processed ("SMI count" obtained via an MSR read of MSR_SMI_COUNT) and the time of the entrance to the function from RDTSC. For `scheduler_tick`, we logged the CPU number, the SMI count, and the timestamp from RDTSC. We extracted our traces with the SystemTap utility. [9] In post-processing, we calculated the deltas between successive handlings of the timer ticks. (See Figure **3**).

Our regular timer tick scenario has a timer tick every millisecond. We generated SMIs using the chipset timer for the short but frequent scenarios and the *Blackbox SMI* method for the hybrid and long SMI scenarios. Our test system was an Intel DQ67SW board running native Centos and the 3.1.4 Linux kernel.

Because the timer interrupt takes precedence over executing code, whether the CPU is idle or busy does not impact the regularity of the regular timer ticks. For this reason, we depict only the idle CPU data in this section. After establishing a baseline with no regular SMIs, we measure the effect of short but frequent SMIs using the *Chipset-based* SMI generation. Following this, we utilize a *Blackbox SMI* scenario of a batch of 8 5*ms* SMIs, once a second to represent an SMM RIMM that takes 40*ms* per second to do integrity measurements using a time-sliced approach.

To analyze the data, we narrow our focus to the deltas between successive invocations of `scheduler_tick` to highlight SMI-caused delays. Numerous short SMIs cause jitter in the timer interrupt handling. Since SMIs take precedence over timer interrupts, the deltas between successive timer interrupts depart from the expected 1*ms*. Deltas greater than 1*ms* occur due to an SMI firing when a timer interrupt would have taken place (Table 1). The delay in timer interrupt handling results in the greater than 1 *ms* delta, that in turn results in the next timer interrupt occurring after less than 1*ms*.

Table 1 shows a small sample of the jitter in the handling of timer interrupts. This effect eventually dissipates, but occurs again as the timer interrupt and SMI occurrences coincide. Even when regular SMIs are short, they can happen to occur at precisely when the timer interrupt fires, resulting in a period of irregular timer interrupts for the short but frequent SMI scenario. Figure 2 depicts this effect.

For the *Blackbox SMI* scenario of a batch of eight 5*ms* SMIs a second, when the batch concludes, execution returns back to the operating system until another SMI occurs. In this scenario and a longer blackbox SMI scenario with a 104*ms* SMI, the privileged software suffers significant portions of time where no forward progress can be made.

These results show that both long and short SMIs can preempt the timer interrupt with different patterns. The short but frequent scenario caused periods of jitter in timer interrupt handling. The long SMI scenarios showed that user and kernel tasks are completely frozen for extended periods of time and a number of timer ticks were missed.

B. *Timer Interrupts with Xen*

To examine the effects of SMIs on timer interrupts in a virtualized environment, we repeated the measurements with a Xen HVM Linux guest running under Xen 4.1.2. The results show that running a virtualized guest introduces a small degree of jitter in the regularity of the

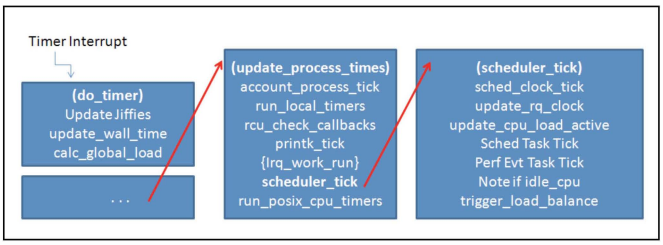


Figure 1 Timer Interrupt Code Flow

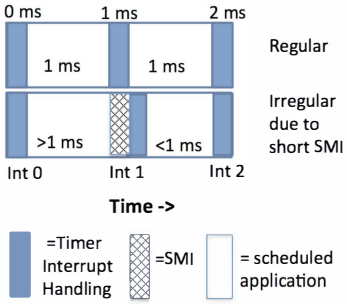
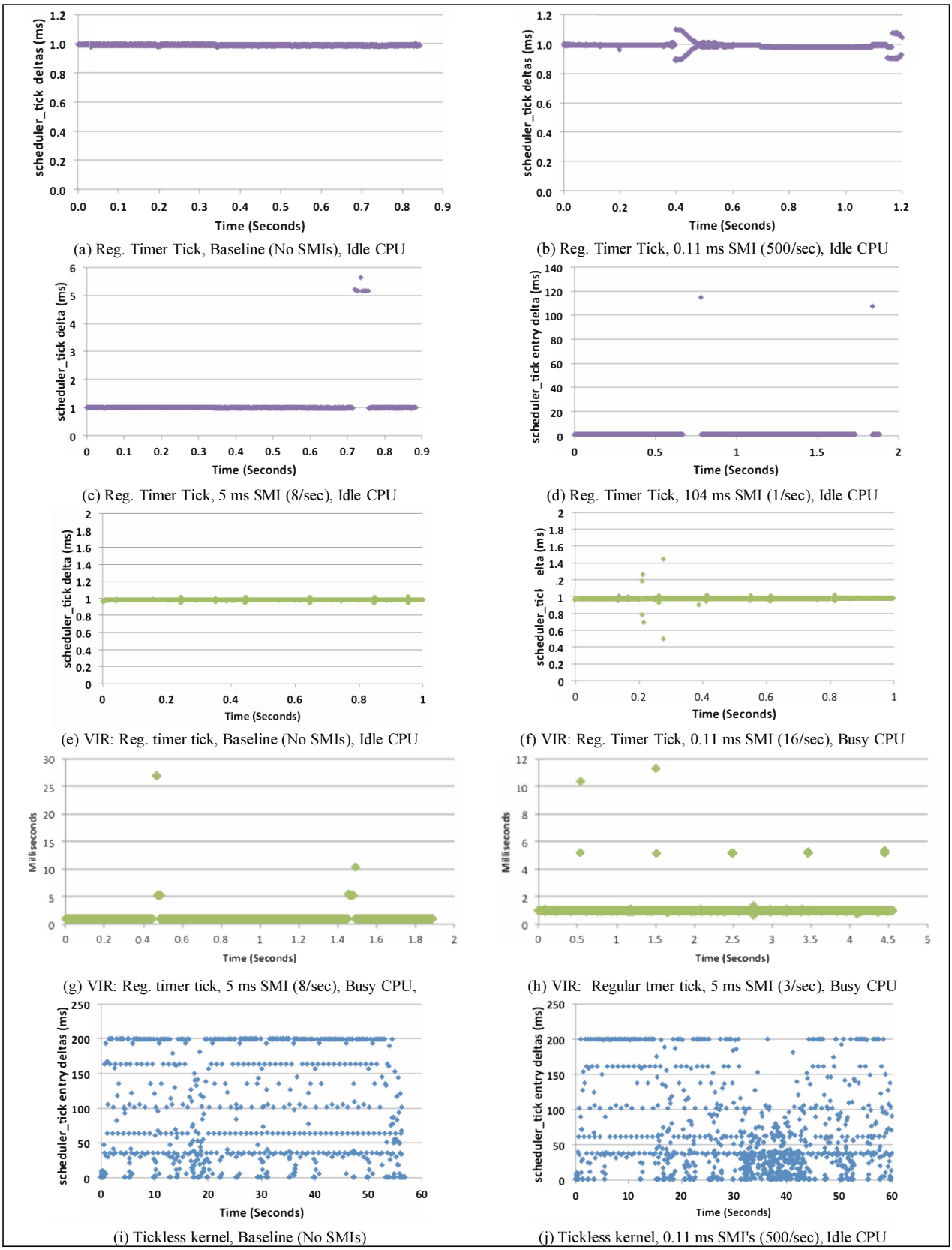


Figure 2 SMI Pre-emption of Timer Interrupt Handling

Table 1 SMI Occurrences and Timer Interrupts

CPU #	SMI Count	Delta (ms)	Location	Notes
	39,089		do_timer	
0	39,090	0.08	scheduler_tick	
1	39,090	0.00	scheduler_tick	
	39,090	0.92	do_timer	<1ms
2	39,090	0.00	scheduler_tick	
0	39,090	0.00	scheduler_tick	
5	39,090	0.00	scheduler_tick	
7	39,090	0.00	scheduler_tick	
	39,091	1.07	do_timer	>1ms
6	39,091	0.00	scheduler_tick	
0	39,091	0.00	scheduler_tick	
7	39,091	0.00	scheduler_tick	
	39,091	0.92	do_timer	<1ms



(a) Reg. Timer Tick, Baseline (No SMIs), Idle CPU

(b) Reg. Timer Tick, 0.11 ms SMI (500/sec), Idle CPU

(c) Reg. Timer Tick, 5 ms SMI (8/sec), Idle CPU

(d) Reg. Timer Tick, 104 ms SMI (1/sec), Idle CPU

(e) VIR: Reg. timer tick, Baseline (No SMIs), Idle CPU

(f) VIR: Reg. Timer Tick, 0.11 ms SMI (16/sec), Busy CPU

(g) VIR: Reg. timer tick, 5 ms SMI (8/sec), Busy CPU,

(h) VIR: Regular tmer tick, 5 ms SMI (3/sec), Busy CPU

(i) Tickless kernel, Baseline (No SMIs)

(j) Tickless kernel, 0.11 ms SMI's (500/sec), Idle CPU

Figure 3: SMI Pre-emption of Timer Interrupt Handling. Scheduler_tick entry deltas for: kernel with regular timer tick (a-d); virtualized with regular timer tick (e-h); and tickless kernel (i-j).

handling of timer interrupts, and adding SMIs perturbs the regularity further. For groups of long SMIs (e.g. groups of eight 5 *ms* SMIs), the guest can experience a significantly longer loss of control which coalesces multiple pre-emptions into one longer one. For example, the Xen HVM guest experiences prolonged losses of control that exceed the 5 *ms* SMI in the range of 10 and 26 *ms* (Figure **3**). We suspect that these increased delays are the effect of SMIs acting upon the virtual machine manager's scheduler which is resulting in the virtual machine not handling the interrupt for a longer period of time and amplifying the impact of shorter SMIs in virtual environments.

C. Tickless Kernel and CPU Power States

When the CPU is busy, the tickless kernel behaves like the regular timer tick, since no ticks are "skipped." During idle periods, however, the tickless kernel can experience large gaps between successive entries into the `scheduler_tick` function (e.g. up to ~200*ms* based on our measurements.) Therefore, we focus here on the idle CPU case. We expect regular SMI activity to subvert the tickless kernel's energy savings, by waking up the CPU to enter SMM. To test this, we gathered data on the processor C-state utilizations using Turbostat. [6] Turbostat produces a log of what percentage of time the processor threads were in a given C-state. [C-states represent incremental power-saving states from C0 (max) to C6 (idle).] We started Turbostat, let the system idle for several seconds, then enabled SMIs, waited a few seconds, disabled SMIs, and ended Turbostat.

In Figure **3**, we show the baseline case for the tickless kernel without SMIs. The timing of the `scheduler_tick` entries varies widely as the kernel avoids unnecessary wake-ups to achieve power savings. The bottom graph shows the results for 500 SMIs/second. It is not readily apparent from the graph if a timer interrupt has been delayed or the kernel was simply idle for a long period of time. To look more closely, we must examine the raw trace data (see Table 2). This shows that the kernel sleeps through the SMI activity as indicated by the increasing SMI count during long periods of kernel idleness. The tickless kernel adaptive timer mechanism is unaware of SMIs and while the kernel is idle, the CPUs transition in and out of SMM processing SMIs. Fortunately, the Linux kernel (since version 2.6.19) has a mechanism to avoid missing jiffy updates due to lost timer ticks by determining how many timer ticks were missed (*ticks)* and incrementing the jiffy count accordingly in `do_timer`. Without such a mechanism, jiffy updates would be lost. The results of our instrumentation (Table 3) show that the `do_timer` function increments the *ticks* value after receiving control following an SMI. When an SMI preempts the kernel for five *ms*, the kernel determines that five timer ticks were missed and sets the *ticks* value accordingly and adds that value to the jiffies count. When our group of eight SMIs concludes, our instrumentation shows the SMI count staying steady and the *ticks* value returning to one as the SMIs subside.

The kernel remained idle through the SMIs, however the CPU was actively processing SMIs. If we limited our analysis to our tickless kernel instrumentation, we would miss a large amount of activity on the system. The kernel instrumentation correctly indicates that there were long periods of idle in the kernel which traditionally would correlate to the CPUs ability to transition into deeper sleep states. However, with SMM RIMMs, regular SMIs are also occurring which would keep the CPU active.

Figure 4 shows that SMIs bring the CPU out of the lowest power C6 state and into the higher power-consuming C0 and C1 states. The short but frequent scenario results in more time spent in higher power C-states than the hybrid scenario that has longer SMIs.

TABLE 2 TICKLESS KERNEL AND 500 SMIS/SECOND

SMI Count	23,351	23,382	23,433
scheduler_tick delta (*ms*)	40	62	102

TABLE 3 `DO_TIMER` TICKS MECHANISM

SMI Count	Ticks	Delta (ms)
19,082	1	1.00
19,083	5	5.21
19,084	5	5.15
…	…	…
19,090	1	0.63
19,090	1	1.00

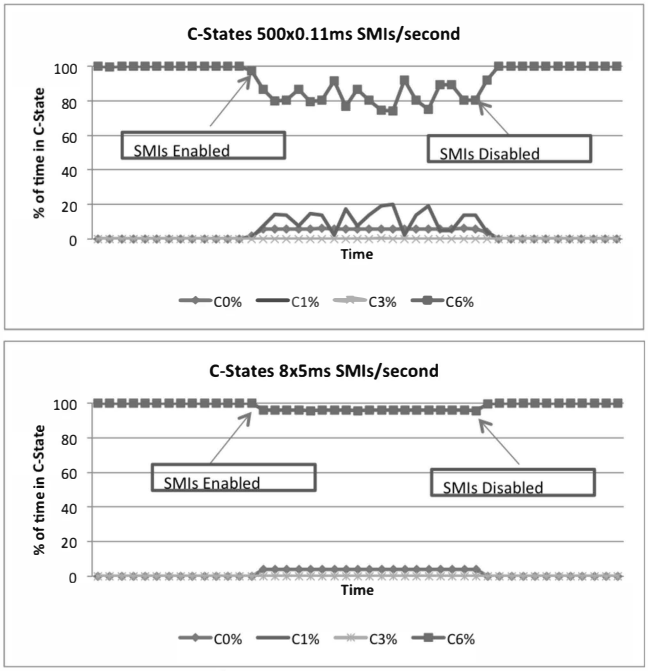


Figure 4 C-States and SMIs

TABLE 4: WARNINGS RECEIVED WITH SMM DELAYS

SMM time (*ms*)	Warning
1.43	`ALSA sound/usb/pcm.c:1213 delay: estimated 144, actual 0`
5 - 999	`ALSA sound/usb/pcm.c:1213 delay: estimated [336 to 384], actual 0`
1000	`ALSA sound/usb/endpoint.c:391 cannot submit urb (err = -27)`

D. Device Driver Impacts

During the previous testing, we noticed a potential problem with USB audio. To study this effect we chose one representative example, a Linux driver for USB speakers, run with a set of Logitech S-150 USB speakers and Linux 3.7.1 kernel on Centos 6.0. We booted into the GUI and began playing a streaming audio file from YouTube. While playing the audio file, we generated progressively longer SMIs using our modified BIOS approach while checking the system log.

USB audio relies upon careful synchronization to keep the audio playback in sync. In our measurements, SMIs perturbed the delay mechanism used by the PCM code and generated warnings starting with our lowest duration SMI and continuing up to 1 second (see Table 4). The warning results from the `snd_pcm_delay` function which defines the playback delay as "the overall latency from the write call to the final DAC." [1] The code provides a warning when the delay estimate is off by more than 2 *ms*. At 1 second preemption, the USB speaker audio stopped.

USB traces gathered using usbmon [31] showed gaps in the USB activity that corresponded very closely with the length of the generated SMI indicating that SMIs preempted USB activity until after the SMI terminated. While we noticed warnings generated at the level of SMI preemptions anticipated for SMM RIMMs, actual driver errors occurred significantly higher than this level. However, the user experience must also be considered and in an equivalent test in Windows using a Microsoft LifeChat LX-3000 USB headset, we subjectively noticed clear audio distortions at 20*ms* SMIs.

E. Linux Process Accounting Anomaly

While preparing for OpenSSL SHA512 benchmarks in our virtualized Centos 6.3 HVM Xen guest with 2.6.32 kernel, we noticed an unexpected phenomenon: When we increased the duration of the SMIs using our modified BIOS, the reported throughput didn't decrease correspondingly. We also noticed that OpenSSL's reported computation time decreased as we generated longer SMIs. The workload reports throughput in bytes per second processed by determining how many computations it did and how much time they took. The OpenSSL benchmark sets up a signal (SIGALRM) for three seconds in the future and performs computations until the signal arrives. The workload reported that the computations took 2.74 seconds when we used 100 *ms* SMIs but the expected 3 seconds when no SMIs occurred. Our examination of OpenSSL showed that it used the Linux kernel's `times` function which reports the amount of user time, system time, child user time, child system time used by a process.

Our measurements using a more recent kernel (3.7.6) showed different behavior. This configuration reported that the OpenSSL benchmark was computing for the full 3 seconds both when 100 *ms* SMIs were enabled and when they were disabled. depicts the scaling of times billed to the application for varying durations of SMIs for the two kernel versions.

To root cause the discrepancy, we instrumented the two Linux kernels to log the flow of time-keeping data used by the `times` and accompanying functions. We also added a tracepoint in the OpenSSL application to capture the time when the signal handler function was called in the application and two tracepoints before and after the computations began in the benchmark.

We started SystemTap to monitor key variables in the kernel functions responsible for the reported process time statistics: `do_sys_times`, `thread_group_times`, `thread_group_cputime`, `task_sched_runtime`, `do_task_delta_exec`, and `scale_utime`. We then started an OpenSSL benchmark run using "openssl speed sha512", with the February 9, 2013 code snapshot of SystemTap. This test allowed us to compare the reported amount of time billed to the process with the tracepoints gathered in the application using the CPU's TSC.

The results show that the SIGALRM signal was received after three seconds in both kernel versions. For the 2.6.32 kernel, this highlighted the discrepancy between the amount of application time as measured by the TSC and the kernel. One kernel function explained the discrepancy: `scale_utime`. This is used to reduce over or under-counting of user or system time due to the point in time when the user or system task was actually interrupted. The code scales the operating system timer tick-based values against the scheduler's record of total runtime. With a 100 *ms* SMI per second, this function increased the billed user time by ~10% which compensated for the loss of time spent in SMM, while for the no SMI scenario, the user time wasn't scaled accordingly. Table 5 shows the scaling calculation for the first three second OpenSSL measurement. As the 2.6.32 kernel didn't call this function in the `do_sys_times` codepath, the user time wasn't adjusted to include the 10% of time spent in SMM leaving the billed process times lower in the 2.6.32 kernel.

TABLE 5 USER TIME SCALING IN SCALE_UTIME - 3.7.6 KERNEL

Variable	Notes	No SMIs	100 *ms* SMI/sec
utime	Unscaled user time	3,003	2,708
rtime	Scheduler's sum_exec_runtime	3,008	3,002
total	User + system	3,009	2,709
[Scaled user time]	(rtime * utime) / total	3,002	3,000

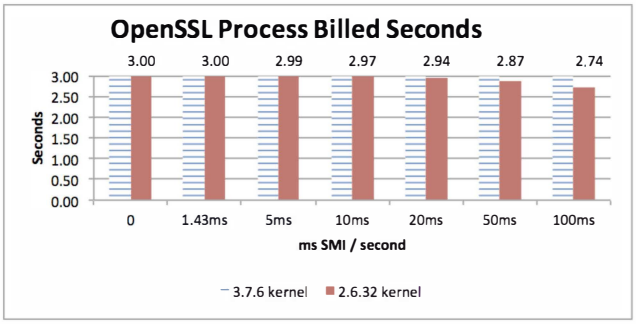


Figure 5 Time Billed to OpenSSL Benchmark on Xen HVM Centos 6.3 guest

E. Discussion

Our examination of disruptions to the regularity of the `scheduler_tick` shows several important effects. In some cases with short but frequent scheduling, SMIs can resonate with the timer interrupt resulting in extended periods of time where the timer interrupt handling may occur late relative to a regular time tick. This may result in timer interrupt handlers closer together or further apart than traditionally done. Additionally, with the longer SMI scheduling option, SMIs that exceed the length of the timer interrupt will cause timer interrupts to be missed. However, the kernel can keep its internal jiffy count accurate. With long SMIs, there can be long periods of time between entries into the process scheduling function. Virtualized environments may experience longer delays as multiple shorter delays coalesce into longer delays. Applications may experience longer wait times since the OS scheduler cannot run.

In the case of an idle tickless kernel, determining if a timer interrupt was delayed or lost due to an SMI is not as straightforward. Our results show that the kernel remained idle while SMIs were occurring which is expected since the kernel is unaware of the loss of control due to SMIs. The C-state analysis showed that while the kernel was idle, the CPU's power utilization was affected by the SMI activity. The short but frequent SMI scheduling scenario resulted in the CPU running in higher power C-states due to frequent wakeups from SMIs that circumvent the power-saving processor modes.

The Linux kernel source code contains assumptions about SMI durations in several places. For example, the function that calibrates the CPU's TSC during boot `native_calibrate_tsc`, uses the `tsc_read_refs` function which has special handling of SMI disturbances. `tsc_read_refs` checks two close reads of the CPU's timestamp counter to ensure that they are less than the declared SMI_THRESHOLD=50000 (CPU clocks) to avoid a scenario where an SMI occurs between the two reads. If the system cannot obtain two close reads of the TSC of a duration less than the SMI_THRESHOLD, it will try up to five times before returning. Prolonged or inopportune SMIs could result in a situation where the TSC couldn't be used as the clocksource for timing due to an inability to properly calibrate it. Other clocksource calibration sections of the Linux kernel feature similar concerns over the impact of an SMI hitting during calibration including functions `pit_calibrate_tsc` and `hpet_next_event`.

Our results raise the question of how the operating system should account for process times when there is prolonged SMI activity on the system: include any SMI times with the billed process time using the times mechanism, or leave this time out of the billed amount? There are benefits and drawbacks to both approaches. Reporting time inclusive of SMI times has the drawback of charging applications for time spent outside of their process which could penalize some applications more than others depending on when the SMIs happened to occur. In our study this resulted in all three seconds being attributed via the `times` mechanism to a process without discarding the portion of the time spent in SMM. The exclusion of SMI times in process time accounting can lead to discrepancies as well. In the case of OpenSSL running on the 2.6.32 kernel, the workload concluded after three seconds based on the CPU's TSC, however the process only believed that it had used 2.74 seconds when 100 *ms* SMIs were active. When SMIs were infrequent and had short durations, their effect on process accounting could essentially be overlooked. For environments that are sensitive to accurate billing of time to users such as cloud providers, new mechanisms are required to more accurately account for the amount of time consumed by long SMIs. However, resolving the fundamental issues in process time accounting will require kernel changes and possible SMM RIMM involvement.

Our analysis of system level SMM effects shows several negative impacts from prolonged SMM time. While certain sections of the Linux kernel have special handling for SMI occurrences, other sections could have differing behavior upon experiencing prolonged SMI durations. Software advances such as tickless kernels, while implemented for other reasons, increase the tolerance of SMM preemptions. Our detailed results demonstrate that systems can spend longer in SMM than current guidelines, however, there are problems that arise at durations below those contemplated for SMM RIMMs.

We showed that SMIs cause periods of timer-interrupt jitter in the short but frequent scenario and extended periods of delays for the longer SMI scenarios. These impacts delay handling of timer interrupts and postpone work on all cores until the SMI completes. Additionally, in an environment where power savings are of increasing importance, SMM RIMMs would bring a reduction in the amount of time CPUs can remain in low power states. In an extreme SMI preemption, we showed a device driver that failed because it interpreted the delay as unresponsive hardware.

VI. Application Level Effects

Because even slight delays can have a perceptible impact on applications, we designed a study to investigate the impact of prolonged SMIs on several types of workloads. The correlation of application and noise granularity [5] is quite relevant to the SMI-based perturbation investigation as SMIs could be long or short, frequent or infrequent, occur regularly or irregularly.

A. Kernel Compilation

Linux kernel compilation involves several key aspects of platform performance including CPU operations, disk I/O, and memory accesses. We used Xen 4.2.1 with a Centos 6.3 Domain 0, and a Centos 6.3 HVM guest with one virtual CPU and two GB of RAM.

The results (Figure 6) show increases in total compilation time that very closely match the level of SMM preemption. Taking 10% of the CPU cycles away for the 100 *ms* SMI scenario resulted in a 10.8% increase in the duration of the kernel compilation.

B. Microbenchmarks with Xen

To examine system performance impacts on a broader set of system activities, we ran two sets of benchmarks, one for Xen's Domain 0 (Xen 4.1.2) and one for a Centos

6.0 HVM guest (Figure 7). We compared how throughput scaled against the baseline for varying levels of SMIs using our modified BIOS setup. For our workloads, we used RC5-72 [12], a compute-intensive workload that brute-forces cryptographic keys (tested on Domain 0 only); Netperf 2.5 for TCP transmit [18] using a gigabit Ethernet device; and XDD for 128KB sequential disk reads using an Intel X25-M SSD. [28]

The left-most bar in the chart shows the percentage of CPU time available to the system after subtracting the amount of time spent in SMM per second. The individual benchmarks all experienced throughput degradations that closely match the amount of CPU cycles taken away for SMIs. With these workloads and long SMIs, SMM latency cannot be hidden by the application as it comes at the cost of performing I/O operations or computations.

By comparison, short but frequent SMI scheduling (bottom of Figure 7) can maintain baseline throughput for some workloads even as the amount of available CPU time decreases. SMI usages that are able to interleave SMIs with I/O processing may be able to avoid the full penalty of the SMI by processing their SMM work in multiple smaller units. The benchmarks didn't experience any stability issues, which helps allay concerns regarding application or I/O device impact at these levels of SMM preemption.

C. *Latency Sensitive Application: Unreal Tournament*

As the USB testing indicated, latency sensitive applications can be problematic for SMM RIMMs. To investigate this further we used Windows Server 2012 and the Unreal Tournament 3 benchmark utility (UTBench) to measure game frame rates. We used our modified BIOS setup for these tests and show the results in Table 6. Although the average frame rates were above 50 fps for all durations but the 495 *ms* SMI, at SMI durations of 20 *ms* and higher the frame rates dipped below 30 frames per second, which is in the range of the user's perception. The finer-grained analysis shows that 20 *ms* delays only dropped below 30 frames per second 0.92% of the time which we didn't notice subjectively however at 50 *ms* delays, the system achieved below 30 frames per second 5.99% of the time which was visually apparent. This latency-sensitive application showed clear sensitivities between the 20 and 50 *ms* SMI durations.

D. *Discussion*

Our results show that application level impacts from SMM time vary based on the characteristics of the application as well as SMI scheduling. Some usages (e.g. compilation) experience degraded throughput while others such as Unreal Tournament and audio playback, are particularly sensitive to long duration SMIs as the user experience is severely degraded. We note that the latency sensitive applications we examined suffered user perceptible impacts at some of the SMI durations proposed for SMM RIMMs.

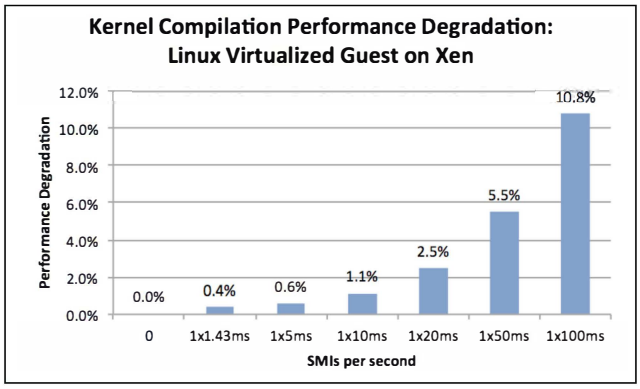


Figure 6: Kernel Compilation Performance for Linux/Xen

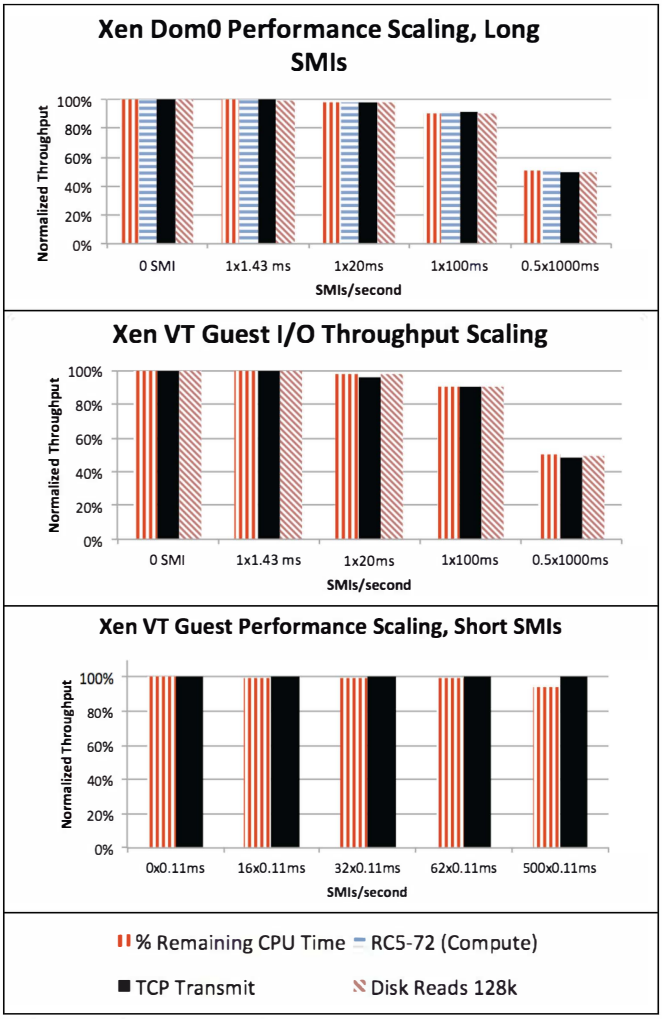


Figure 7: Xen Microbenchmarks

TABLE 6 UNREAL TOURNAMENT FRAME-RATE BINNING

MS/smi	0-5	5-10	10-15	15-20	20-25	25-30	30-35	35-40	40-45	45-50	50-55	55-60	60+
0	0	0	0	0	0	0	0	0	0	0	0	0	100
1.43	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.08	99.91
5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.09	0.16	0.15	99.60
10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.87	0.12	0.14	0.02	98.79
20	0.00	0.00	0.00	0.00	0.00	0.92	1.59	0.37	0.17	0.05	0.00	0.00	96.90
50	0.00	0.00	0.00	5.99	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	94.00
99	0.00	10.83	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	89.17
495	50.24	0.00	0.04	0.03	0.02	0.02	0.00	0.00	0.00	0.00	0.01	0.00	49.64

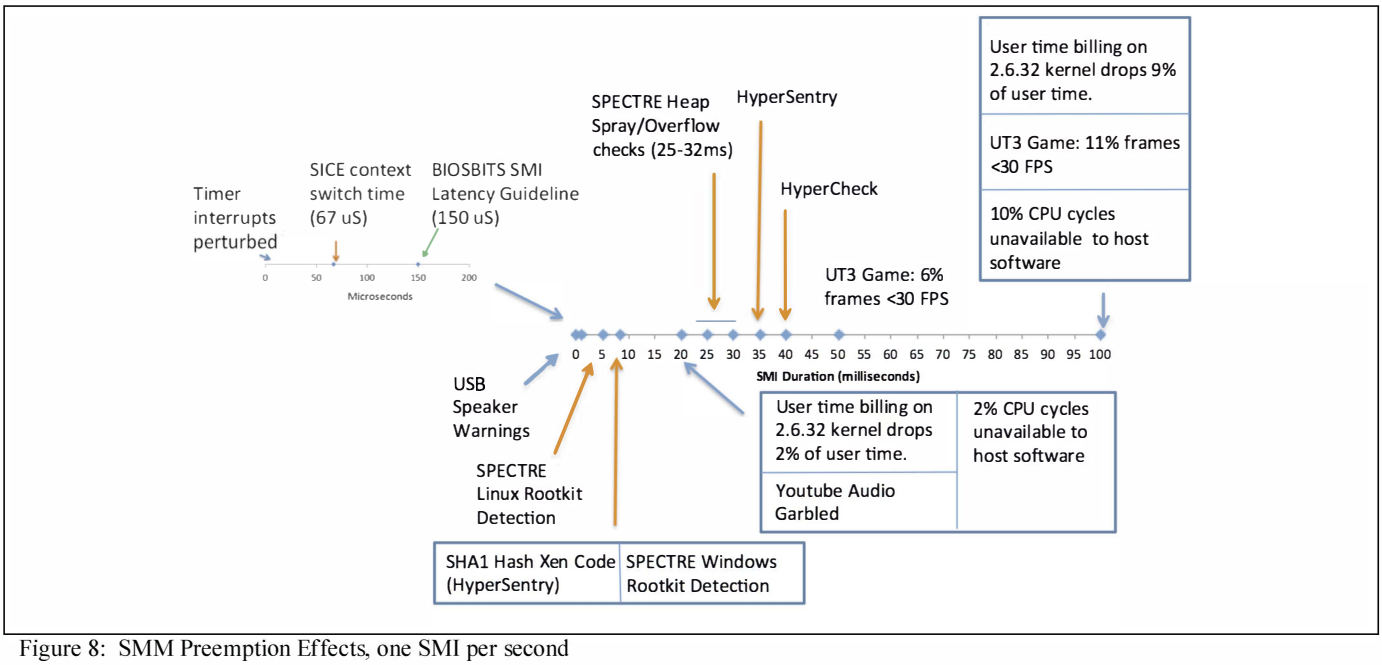


Figure 8: SMM Preemption Effects, one SMI per second

VII. Conclusions and Future Work

In this study, we examined the impact of System Management Mode on system and application performance. Our results show that time spent in this mode causes warnings, perceptible degradations for latency-sensitive applications, throughput impacts, delays, and inaccurate time accounting at the hypervisor, kernel, and application levels (Figure 8).

The motivation for our study was a series of proposals for SMM RIMMs, a novel way to improve detection of malware in privileged software. The unique ability of an SMM RIMM to reside in a protected location and provide confidence that the privileged software isn't compromised is extremely valuable. However, our results show that the three published SMM RIMM approaches that we surveyed [3][25][29] will cause a range of unacceptable side effects.

For SMM repurposing to succeed, either the specific approaches must be changed to reduce the SMM duration; or the runtime software stack must undergo a redesign to increase tolerance or response to the "missing time" that results from SMIs. Shorter but frequent SMIs can remedy some of the effects but will still take the same amount of time away from CPU-intensive applications, result in interrupt-handling jitter, and sacrifice the atomicity of the security checks allowing malware an increased ability to hide. Redesigning the software stack require a shift in current thinking about the ability of privileged software to control the system. SMM RIMMs break both kernel and hypervisor assumptions of platform control as the most privileged kernel or hypervisor code can be preempted by an SMI.

References

[1] ALSA http://www.alsa-project.org/alsa-doc/alsa-lib/group___p_c_m.html, accessed July 30, 2013.

[2] AMD, AMD64 Architecture Programmer's Manual, Volume 2: System Programming.

[3] A. Azab, P. Ning, et al., "HyperSentry: enabling stealthy in-context measurement of hypervisor integrity," CCS. Chicago, IL, 2010.

[4] A. Azab, P. Ning, and X. Zhang, "SICE: A Hardware-Level Strongly Isolated Computing Environment for x86 Multi-core Platforms," CCS. Chicago, IL 2011.

[5] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj,: "Benchmarking the effects of operating system interference on extreme-scale parallel machines," Cluster Computing 11(1): pp. 3-16, 2008.

[6] L. Brown,: Linux Idle Power Checkup, https://events.linuxfoundation.org/slides/2010/linuxcon2010_brown.pdf, accessed July 31, 2013, 2010.

[7] Camden Associates, "Device Driver and BIOS Development for AMD Multiprocessor and Multi-Core Systems,"Advanced Micro Devices, Inc. 2006.

[8] L. Duflot, O. Levillian, B. Morin, and O. GrumeLard, "Getting into SMRAM: SMM reloaded," CanSecWest. Vancouver, Canada, 2009.

[9] F. Eigler, V. Prasad, et al.: "Architecture of systemtap: a Linux trace/probe tool,"http://sourceware.org/systemtap/archpaper.pdf, accessed July 31, 2013, 2005.

[10] K. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE Press: 1-12. Austin, Texas, 2008.

[11] R. Ghosh-Roy, "Disabling SMIs on Intel® ICH5 Chipsets," http://www.mathworks.com/matlabcentral/fileexchange/18832-disabling-smis-on-intelr-ich5-chipsets, accessed July 31, 2013.

[12] B. Hayes, "Collective Wisdom," American Scientist, 1998.

[13] A. Ugal, "Hard Real Time using Xenomai on Intel Multi-Core Procesors", white paper, Intel, 2009.

[14] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual (Vol. 3).

[15] Intel, Intel Itanium Architecture Software Developer's Manual, Revision 2.3. Volume 2: System Architecture.

[16] K. Mannthey, "System Management Interrupt Free Hardware," IBM Linux Technology Center. http://linuxplumbersconf.org/2009/slides/Keith-Mannthey-SMI-plumers-2009.pdf, accessed July 31, 2013.

[17] W. Mauerer. Professional Linux Kernel Architecture, p. 909. Wrox, 2008.

[18] Netperf, http://www.netperf.org, accessed July 31, 2013.

[19] F. Petrini, D.J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," SC 2003. Phoenix, Arizona , 2003.

[20] S. Siddha, V. Pallipadi, and A. Van De Ven, "Getting maximum mileage out of tickless," Proceedings of the Linux Symposium, Ottawa, Canada, 2007.

[21] P. Stultz, "System and method for processing system management interrupts in a multiple processor system," U.S. Patent 7,200,701 B2, issued April 3, 2007.

[22] J. Triplett and B. Triplett, "BITS: BIOS Implementation Test Suite,"http://www.linuxplumbersconf.org/2011/ocw/system/presentations/867/original/bits.pdf

[23] D. Tsafrir, Y. Etsion, D. Feitelson, and S. Kirkpatrick, "System noise, OS clock ticks, and fine-grained parallel applications." Proceedings of the 19th annual international conference on Supercomputing. Cambridge, Massachusetts, ACM: 303-312 , 2005.

[24] J. Wang, K. Sun, and A. Stavrou, "An analysis of system management mode (SMM)-based integrity checking systems and evasion attacks," Technical Report, George Mason University, GMU-CS-TR-2011-8 (2011)

[25] J. Wang, A. Stavrou, and A. Ghosh, "HyperCheck: a hardware-assisted integrity monitor," Lect. Notes Comput. Sci. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 6307 LNCS: 158-177, 2010.

[26] C. Williams, "Realtime in the Enterprise," RTLWS11, Dresden, Germany,https://www.osadl.org/fileadmin/dam/presentations/RTLWS11/clarkw-realtime-in-the-enterprise.pdf, accessed: July 30, 2013, 2009.

[27] R. Wojtczuk, "Subverting the Xen Hypervisor," Black Hat, Las Vegas, Nevada, 2008.

[28] XDD, http://sourceforge.net/projects/xdd

[29] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "SPECTRE: A dependable Introspection Framework via System Management Mode," DSN, Budapest, Hungary, 2013.

[30] "Configuring and Tuning HP ProLiant Servers for Low-Latency Applications", white paper, HP, 2013.

[31] P. Zaitcev, "The usbmon: USB monitoring framework," Red Hat, http://people.redhat.com/zaitcev/linux/OLS05_zaitcev.pdf, accessed July 31, 2013.