I/O Processing in a Virtualized Platform: A Simulation-Driven Approach

Vineet Chadha
Renato J. Figueiredo
Advanced Computing and Information
Systems Laboratory, University of Florida
{chadha,renato}@acis.ufl.edu

Ramesh Illikkal, Ravi Iyer
Jaideep Moses, Donald Newell
Intel Corporation, Hillsboro, OR
{ramesh.g.illikkal, ravishankar.iyer,
jaideep.moses, donald.newell}@intel.com

Abstract

Virtualization provides levels of execution isolation and service partition that are desirable in many usage scenarios, but its associated overheads are a major impediment for wide deployment of virtualized environments. While the virtualization cost depends heavily on workloads, it has been demonstrated that the overhead is much higher with I/O intensive workloads compared to those which are compute-intensive. Unfortunately, the architectural reasons behind the I/O performance overheads are not well understood. Early research in characterizing these penalties has shown that cache misses and TLB related overheads contribute to most of I/O virtualization cost. While most of these evaluations are done using measurements, in this paper we present an execution-driven simulation based analysis methodology with symbol annotation as a means of evaluating the performance of virtualized workloads. This methodology provides detailed information at the architectural level (with a focus on cache and TLB) and allows designers to evaluate potential hardware enhancements to reduce virtualization overhead. We apply this methodology to study the network I/O performance of Xen (as a case study) in a full system simulation environment, using detailed cache and TLB models to profile and characterize software and hardware hotspots. By applying symbol annotation to the instruction flow reported by the execution driven simulator we derive function level call flow information. We follow the anatomy of I/O processing in a virtualized platform for network transmit and receive scenarios and demonstrate the impact of cache scaling and TLB size scaling on performance.

Categories and Subject Descriptors C.4 [**Performance of Systems**]: Design studies, Measurement techniques, performance attributes.

General Terms Measurement, Performance

Keywords Simulation, Virtualization, Performance model, Xen, Virtual Machines

1. Introduction

In recent years, virtualization has re-emerged as a means to improve the utilization of available compute resources and to enhance the overall system reliability [22]**.** While reliability and cost arguments motivate the use of virtualized systems, the overhead due to virtualization is a major obstacle for widespread adoption. It is well understood that compute-intensive workloads suffer low virtualization overheads compared to I/O-intensive workloads.

In order to design I/O-efficient virtualized systems, it is a key challenge to understand how micro-architectural features impact the performance of I/O workloads in virtualized environments. Recent studies [4, 5] show that most of I/O overhead could be attributed to increased cache and TLB misses. While previous studies have relied on measurements to assess the performance impact of I/O virtualization of existing workloads and systems, it is important to understand architectural-level implications to guide the design of future platforms and the tuning of system software for virtualized environments.

This paper addresses this problem and presents a simulation-based analysis methodology which extends a full system simulator with symbol annotation of the entire software stack in virtualized environments – including the hypervisor, service and guest domains. The key contributions of this paper are twofold: first, we describe methodologies and issues involved in analyzing a virtualized workload on an existing simulator, including symbol annotation to differentiate the various components in the software stack; and second, we demonstrate the feasibility and initial results of using this extended simulation environment to evaluate the profile of cache and TLB misses in a representative I/O workload. Results from this case study shows that we can correlate simulated results with important events across these different components of the stack. To our knowledge, this is the first study using full-system simulation to estimate overheads and profile the anatomy of I/O processing in a virtualized system. Using full-system simulation, we profile the workload following the execution path of network packet handling inside the virtual environment. Furthermore, we perform architecture-level quantitative analyses using cache and TLB simulation models that are integrated with the execution-driven simulation and symbol annotation framework.

We chose to model cache and TLB in detail since the cost associated with these resources are considered to be high. By profiling the execution and collecting architectural data, we show the causes for cache misses as well as TLB misses. We also show the impact of cache size and TLB size on I/O performance by scaling these resources. In this paper we provide a detailed analysis of the current I/O VM architecture of a representative open-source VMM (Xen [20]), using the SoftSDV [26] execution-driven simulator extended with symbol annotation support and a network I/O workload (iperf). We selected network I/O since inter-VM communication and service VM architecture is integral part of the current IO virtualization architecture. Also, recent studies have

VEE'07 June 13–15, 2007, San Diego, California, USA.

indicated that the I/O VM architecture becomes a performance bottleneck as we stress the network I/O throughput [4, 5].

The rest of the paper is organized as follows. The motivation behind the current work is described in Section 2. Section 3 describes the simulation methodology, tools and symbol annotations. Section 4 details the software and architectural anatomy of I/O processing by following the execution path through guest domain, hypervisor and the I/O VM domain. Also, we provide initial results of resource scaling in Section 4. Section 5 describes related work. We conclude with summary and future work in Section 6.

2. Motivation and Background

The present work is motivated by the fact that current system evaluation methodologies for classic and para-virtualized VMs are based on measurements of a deployed virtualized environment on a physical machine. Although such an approach gives good estimates of performance overheads for a given physical machine, it lacks flexibility in determining the resource scaling performance. In addition, it is difficult to replicate a measurement framework on different system architectures. We suggest that it is important to move towards a full system simulation methodology because it is a flexible approach in studying different architectures.

Simulation-based approaches have been extensively used in computer architecture to design and analyze the performance of upcoming system architecture [21, 28, 29]. A simulation-based methodology for virtual environments is also important to guide the design and tuning of architectures for virtualized workloads, and to help software systems developers to identify and mitigate sources of overheads in their code.

A driving application for simulation-driven analysis is I/O workloads. It is important to minimize performance overheads of I/O virtualization in order to enable efficient workload consolidation use models. For example, in a typical three tier data center environment, Web servers providing the external interface are typically I/O-intensive; a low performing front end server could bring the overall data center performance down. It is also important to minimize performance overheads to enable emerging usage models of virtualization. New architecture features could also drive the virtualization evolution. For example, offloading the I/O services to an isolated, specialized I/O domain and communicating to it through messages is motivated by similar arguments that have motivated micro-kernels [27]. Enabling a low latency, high bandwidth inter-domain communication mechanism between VM domains is one of the key architecture elements which could push this distributed services architecture evolution forward.

2.1 Full System Simulator and Performance

Full system simulators are often employed to evaluate design, development and testing on traditional hardware and software for upcoming architectures. There are several cycle-accurate simulators that support the x86 instruction set architecture [18, 21]; in this paper we use the SoftSDV simulator [26] as a basis for the experiments. SoftSDV not only supports fast emulation with dynamic binary translation, but also allows proxy I/O devices to connect a simulation run with physical hardware devices. It also supports multiple sessions to be connected and synchronized through a virtual SoftSDV network. For cache and TLB modeling we integrated CASPER [24] – a functional simulator which offers rich set of performance metrics and protocols to determine cache hierarchical statistics.

2.2 I/O Virtualization in Xen

We briefly look at the various I/O architecture options and in particular the I/O VM model used in Xen 3.0. The design of I/O architecture in virtual systems is often driven by tradeoffs between fault tolerance and I/O performance. In this context, I/O architectures can be broadly divided into split I/O and direct I/O. Direct I/O is generally adopted in classical virtual machines like VMware to boost I/O performance where front end and backend drivers often communicate using system calls. Split I/O architecture, adopted by para-virtualized machines, isolates backend drivers in a separate VM to communicate with front end drivers through inter-process communication (IPC), resulting in an approach similar to those found in micro-kernels.

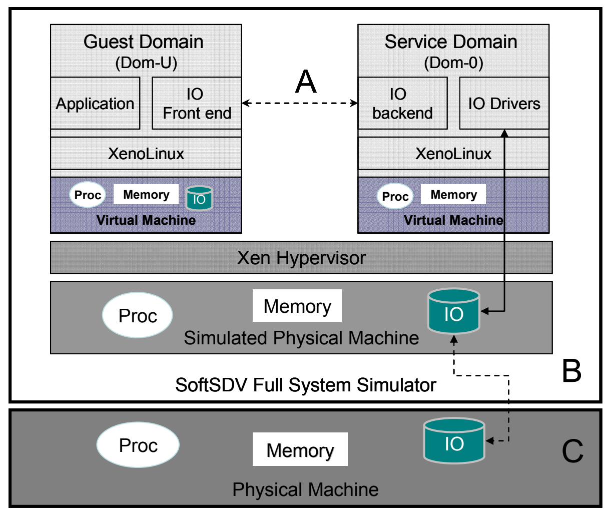


Figure 1: Full system simulation environment with Xen Execution includes (A) Xen Virtual Environment (B) SoftSDV Simulator (C) Physical Machine.

The Xen I/O architecture has evolved from hypervisor-contained device drivers (direct I/O) to a split I/O. The primary goal of the I/O service VM based Xen I/O architecture is to provide fault isolation from misbehaved device drivers. It also enables the use of unmodified device drivers. The Xen network I/O architecture is based on a communication mechanism to transfer information between guest and service VM (Figure 1, (A)). The guest domain's front end driver communicates with backend drivers through IPC calls. The virtual and backend driver interfaces are connected by an I/O channel. This I/O channel implements a zero-copy page remapping mechanism for transferring packets between multiple domains. We describe the IO VM architecture in detail when we present the life-of-packet analysis in Section 4.

3. Analysis Methodology

In this section, we present an overview of Xen as case study using full system simulation analysis methodology. We also show how we identified the flow of packets inside a multi-layer software environment with multiple VMs and hypervisor along with micro-architectural details of the processor events of interest. Figure 2 summarizes the profiling methodology and the tools we used. The following sections describe the individual steps in detail; these include (1) virtualization workload (2) full system simulation (3) instruction trace (4) performance simulation with detailed cache and TLB simulation and (5) symbol annotation.

3.1 Full System Simulation: Xen VMM as workload

The first step in the methodology for getting a detailed understanding of the workload is to run a virtualized environment, unmodified, within a full system execution driven simulator. In the analysis presented in this paper, the Xen virtualized environment includes the Xen hypervisor, the service domain (Dom0) with its O/S kernel and applications, and a guest, "user" domain (DomU) with its O/S kernel and applications (Figure 1). In order to analyze a network-intensive I/O workload, the iperf benchmark application is executed in DomU.

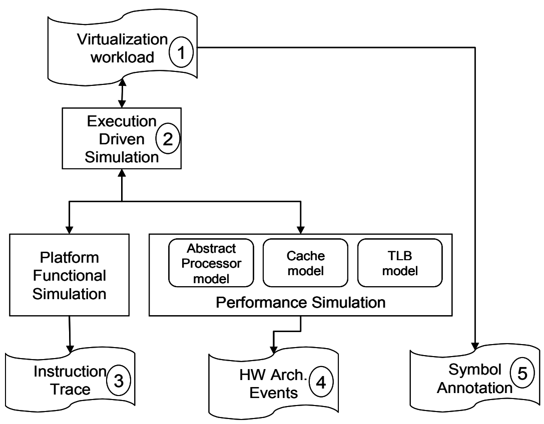


Figure 2: Execution Driven Simulation & Symbol annotated profiling methodology – (1) Virtual workload (2) Execution driven simulation (3) Instruction trace form functional model (4) Hardware events such as cache hit (5) EIP Symbol annotation.

This environment allows us to tap into the instruction flow to study the execution flow and to plug in detailed performance models to characterize architectural overheads. As explained in Section 2.1, the DomU guest uses a front end driver to communicate with a backend driver inside Dom0, which controls the I/O devices. We synchronized two separate simulation sessions to create a virtual networked environment for I/O evaluation. The execution-driven simulation environment combines functional and performance models of the platform. For this study, we chose to abstract the processor performance model and focus on accurate cache and TLB models to enable the coverage of a long period in the workload (approximately 1.2 billion instructions).

3.2 Instruction trace

Functional simulation provides stateless execution of systems instructions; no state is maintained for TLB and cache access. The SoftSDV functional simulator loads and executes the Xen hypervisor and guest images. When iperf executes and communicates with the I/O services in Dom0, the instructions issued by the hypervisor, DomU and Dom0 are decoded and executed by the functional model. This enables tracing of the flow of execution at the instruction level for the entire workload execution which serves as a starting point for the analysis. The instruction trace can then be parsed to identify important events such as context switches and function calls. For example, we mapped the next instruction after the CALL instruction to the symbols collected from hypervisor, application and drivers to obtain a sequence of functions in execution.

3.3 Symbol Annotation

In Linux, symbols for the kernel (and, similarly for applications and drivers) can be located in compile-time files (such as system.map for kernel). Symbols for running process can be collected from proc kernel data structures. In order for us to gain more insight into the packet flow and software modules inside the virtualization software layers, we also added symbol information to the execution flow. Symbols were collected from the Xen hypervisor and guest operating system, and we also added symbols from applications and drivers to complete the instruction trace annotation. The annotation process matches the simulated instruction pointer (EIP in x86) with such symbols, allowing the tagging of regions of the instruction trace (and associated statistics) with code executed by the different components of the virtualized environment. For example, this methodology has allowed us to follow the life of a network packet inside the Xen virtualized environment, which is described in Section 4. An example execution flow after symbol annotation is given in Figure 3. These decoded instructions from the functional model are then provided to the performance model which simulates the architectural resources and timing for the instructions executed.

3.4 Performance Statistics

We collect instruction flow and associated performance statistics from cache and TLB models to identify performance hotspots. We can leverage detailed models of cache and TLB to characterize the impact of cache and TLB size on the I/O virtualization performance. Results from the resource scaling studies for the Xen virtualized environment are provided in Section 4.2. A simulated platform also provides us with the capability of changing the underlying hardware architecture to evaluate architecture enhancements and their impact on workload performance. An example of the execution flow with performance details is given in Figure 4.

EIP	Function Name	Module
ff110760	stop_timer	[hypervisor]
ff1103c0	remove_entry	[hypervisor]
ff123c90	smp_send_event_check_mask	[hypervisor]
ff110650	set_timer	[hypervisor]
ff110540	add_entry	[hypervisor]
ff110370	up_heap	[hypervisor]
ff124570	update_dom_time	[hypervisor]
ff117820	context_switch	[hypervisor]
ff117690	__context_switch	[hypervisor]
ff11d820	write_ptbase	[hypervisor]
ff127810	__copy_to_user_ll	[hypervisor]
ff10f980	do_softirq	[hypervisor]
ff124280	get_s_time	[hypervisor]
ff114120	reprogram_timer	[hypervisor]
ff124280	get_s_time	[hypervisor]
ff1493d5	create_bounce_frame	[hypervisor]
c023da30	evtchn_do_upcall	[kernel]
c0106b60	do_IRQ	[kernel]
c0142d30	__do_IRQ	[kernel]

Figure 3: Annotated instruction flow.

EIP	Function Name	Module	L1 Hits	L1 misses	L1 nst_misses	L1 data_misse	L1 read_miss	L1 write_miss	L1 total refs	L1 Instr refs	L1 data refs	L1 data_rds	L1 data_wrs	L2 Hits	L2 misses	L2 nst_misses	L2 data_misse	L2 read_miss	L2 write_miss	L2 total refs	L2 Instr refs	L2 data refs	L2 data_rds	L2 data_wrs	TLB64dataHit	TLB64dataMis	TLB64InstHitC	TLB64InstMis
ff110760	stop_timer	[hypervisor]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ff1103c0	remove_entry	[hypervisor]	37	2	2	0	0	0	39	25	14	7	7	2	0	0	0	0	0	2	2	0	0	0	14	0	25	0
ff123c90	smp_send_event_check_mask	[hypervisor]	17	0	0	0	0	0	17	12	5	2	3	0	0	0	0	0	0	0	0	0	0	0	5	0	12	0
ff110650	set_timer	[hypervisor]	35	1	1	0	0	0	36	18	18	8	10	1	0	0	0	0	0	1	1	0	0	0	18	0	18	0
ff110540	add_entry	[hypervisor]	49	1	1	0	0	0	50	32	18	9	9	1	0	0	0	0	0	1	1	0	0	0	18	0	32	0
ff110370	up_heap	[hypervisor]	37	1	1	0	0	0	38	22	16	5	11	1	0	0	0	0	0	1	1	0	0	0	16	0	22	0
ff124570	update_dom_time	[hypervisor]	99	3	3	0	0	0	102	58	44	27	17	3	0	0	0	0	0	3	3	0	0	0	44	0	58	0
ff117820	context_switch	[hypervisor]	17	2	2	0	0	0	19	11	8	5	3	2	0	0	0	0	0	2	2	0	0	0	8	0	11	0
ff117690	__context_switch	[hypervisor]	40	2	2	0	0	0	42	29	13	7	6	2	0	0	0	0	0	2	2	0	0	0	13	0	29	0
ff11d820	write_ptbase	[hypervisor]	23	0	0	0	0	0	23	11	12	8	4	0	0	0	0	0	0	0	0	0	0	0	12	0	11	0
ff127810	__copy_to_user_ll	[hypervisor]	32	0	0	0	0	0	32	21	11	6	5	0	0	0	0	0	0	0	0	0	0	0	11	0	21	0
ff10f980	do_softirq	[hypervisor]	34	2	2	0	0	0	36	26	10	7	3	2	0	0	0	0	0	2	2	0	0	0	9	1	24	2
ff124280	get_s_time	[hypervisor]	56	0	0	0	0	0	56	35	21	9	12	0	0	0	0	0	0	0	0	0	0	0	20	1	33	2
ff114120	reprogram_timer	[hypervisor]	39	2	2	0	0	0	41	27	14	10	4	2	0	0	0	0	0	2	2	0	0	0	12	2	27	0
ff124280	get_s_time	[hypervisor]	16	1	1	0	0	0	17	11	6	2	4	1	0	0	0	0	0	1	1	0	0	0	6	0	10	1
ff1493d5	create_bounce_frame	[hypervisor]	36	2	2	0	0	0	38	24	14	9	5	2	0	0	0	0	0	2	2	0	0	0	13	1	24	0
c023da30	evtchn_do_upcall	[kernel]	59	2	2	0	0	0	61	20	41	29	12	2	0	0	0	0	0	2	2	0	0	0	41	0	19	1
c0106b60	do_IRQ	[kernel]	69	5	4	1	1	0	74	52	22	12	10	5	0	0	0	0	0	5	4	1	1	0	17	5	51	1
c0142d30	__do_IRQ	[kernel]	10	1	1	0	0	0	11	7	4	2	2	1	0	0	0	0	0	1	1	0	0	0	4	0	6	1

Figure 4: Performance information added to annotated instruction flow.

3.5 Environment Setup for Virtualized Workloads

The setup and priming of a workload within a simulation environment can be time-consuming. To facilitate the setup for simulation of the virtualized environment, we first create a raw virtual disk which is then ported to the simulator.We chose to apply physical-to-virtual disk conversion as generally it is time consuming to test and commit changes in a simulated medium; creating a disk image outside the simulator facilitates the setup and testing of the workload. Also, for current architectures, decoupling simulation from testing images gives us the flexibility to test O/S or application modifications and execution on a physical machine. For example, even though we executed *iperf* for the experimental evaluation, the above methodology provides flexibility to support any application. To convert a physical disk into a virtual disk, we modified the physical disk partition table to create a miniature replica of the physical disk using the Linux *dd* utility.

Also, to reduce booting time of the installed O/S, we customized a stripped-down version of the physical image by removing unnecessary boot time processes. For guest Xen images, we created a blank virtual disk and populated it with minimal RPM installation packages primarily to facilitate iperf run and networking with Dom0.

The CASPER cache model exports APIs to print or collect the instruction traces during a simulation run. As shown in Figure 5, an instruction parser is used to parse different instruction events such as INT (interrupts, system calls), MOV CR3 (address space switch), and CALL (function call). These traces were dumped into a file with run-time virtual address information, as well as cache and TLB statistics. SoftSDV system call (SSC) utilities facilitate transfer of data between host and simulated guest.

These utilities are important as we gathered run time symbols of kernels and application from the proc kernel data structure to transfer to the host system (for example, /proc/kallsyms for kernel symbols). For iperf runtime symbols, we mapped process ID with corresponding process ID in proc directory. These run-time symbols, in addition to compile-time symbols from kernel, hypervisor, drivers and iperf, provide mapping information between functions and virtual addresses.

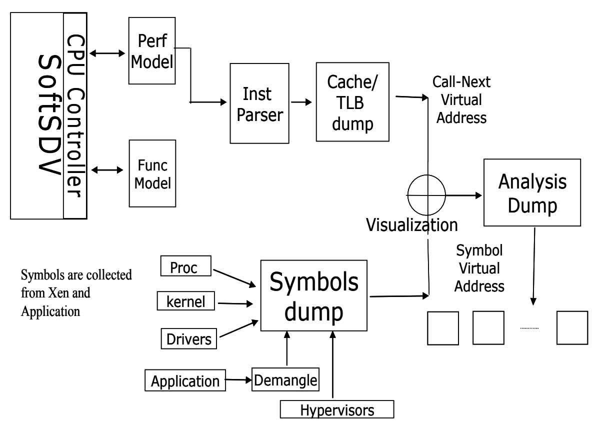


Figure 5: Performance simulation model is used to collect instruction traces of virtualized workload. SoftSDV CPU controller controls execution in performance or functional mode. Instruction traces are parsed and mapped with symbol dumps to create IO call graph

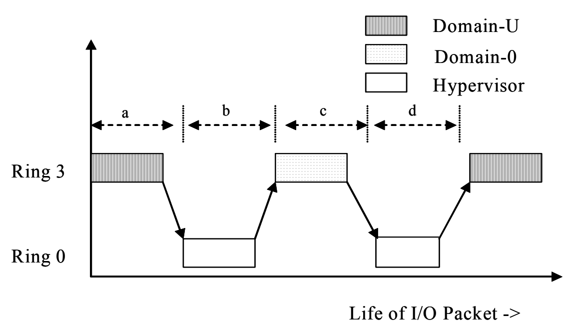


Figure 6: Life of an I/O Packet (a) unprivileged domain (b) Grant table mechanism – switch to hypervisor (c) Timer interrupt to start the context switch (d) Privileged domain.

We annotated symbols to keep track of the source of a function call invocation. Note that there can be duplicate symbols when we sum up all collected symbols into a file. We removed these duplicates and formatted the data collected from proc structures to represent data in a useful way. In some cases, it was necessary to manually resolve ambiguities in virtual address spaces through checkpoint at virtual address during re-run of a simulated SoftSDV session. Linux utilities such as *nm* and *objdump* are often used to collect symbols from compile time symbol tables. In general, any application can be compiled to provide symbol table information. In C++ applications (such as iperf), function name mangling in object code is used to provide distinct name for functions that share the same name. Essentially, it adds some randomness at prefix and suffix of the function name. We used the demangle option of the *nm* utility to obtain the correct function for *iperf* application. Xen kernel and hypervisor symbols are collected from /boot/System.map-2.6.13-xen and $INSTALL/xen/xen-syms. We compared and visualized instruction trace and symbol dump into user-friendly format to obtain call graphs and statistical information such as cache and TLB misses per function invocation.

EIP	Function	Module	Instr count
c02aeb50	do_sock_write	Dom-U	0
c02b2720	lock_sock	Dom-U	78
c0331a00	_spin_lock_bh	Dom-U	85
c02f2590	tcp_current_mss	Dom-U	154
c01e5560	copy_from_user	Dom-U	255
c01e54b0	__copy_from_user_ll	Dom-U	277
c023fa00	__alloc_skb	Dom-U	505
c02b3380	alloc_skb_from_cache	Dom-U	527
c0164d90	kmem_cache_alloc	Dom-U	543
c0164d90	kmem_cache_alloc	Dom-U	591
c01e5560	copy_from_user	Dom-U	1552
c01e54b0	__copy_from_user_ll	Dom-U	1574
c023fa00	__alloc_skb	Dom-U	2029
c02b3380	alloc_skb_from_cache	Dom-U	2051
c0164d90	kmem_cache_alloc	Dom-U	2067
c0164d90	kmem_cache_alloc	Dom-U	2115
c01649b0	cache_alloc_refill	Dom-U	2136
c0331b40	_spin_lock	Dom-U	2170
c01647a0	cache_grow	Dom-U	2217
c0331b40	_spin_lock	Dom-U	2245
c01646e0	kmem_flagcheck	Dom-U	2277
c0163860	kmem_getpages	Dom-U	2290
c01499c0	__alloc_pages	Dom-U	2309
c01498d0	get_page_from_freelist	Dom-U	3581
c01497f0	zone_watermark_ok	Dom-U	3611
c0149550	buffered_rmqueue	Dom-U	3662
c014a140	_page_state_offset	Dom-U	3728
c0148f80	prep_new_page	Dom-U	3754
c01517d0	page_address	Dom-U	3832
c014a170	mod_page_state_offset	Dom-U	3864
c01645f0	alloc_slabmgmt	Dom-U	3923
c0164d90	kmem_cache_alloc	Dom-U	3937
c0164760	set_slab_attr	Dom-U	3999
c0164660	cache_init_objs	Dom-U	4028
c0331b40	_spin_lock	Dom-U	4085
c0331b40	_spin_lock	Dom-U	4140
c0164710	slab_get_obj	Dom-U	4184
c0164710	slab_get_obj	Dom-U	4210
c02b7230	sk_stream_mem_schedule	Dom-U	7810
c01e5560	copy_from_user	Dom-U	9175
c01e54b0	__copy_from_user_ll	Dom-U	9197
c02f2ed0	__tcp_push_pending_frames	Dom-U	9382
c02f2c10	tcp_write_xmit	Dom-U	9396
c02f2750	tcp_init_tso_segs	Dom-U	9417
c02b2780	release_sock	Dom-U	9478

Figure 7: Dom-U call graph: Socket allocation, user-kernel data copy and finally TCP transmit write (period (a) in Figure 6).

4. Experiments and Simulation Results

We conducted experiments in two parts. First, we collected important events such as the occurrence of CALL instructions to determine the flow of a virtual Ethernet packet. Secondly, we executed the iperf application to generate both transmit and receive workloads to perform cache scaling studies. Figure 5 shows the simulation framework implementation to obtain call graph information and perform cache scaling studies. As shown in Figure 5, the CPU controller layer in SoftSDV integrates with a performance or functional model.

The platform configuration for this study is set to a single processor with 2 levels of cache (32 KB first level data and instruction cache, 2MB L2 cache) and with 64-entry instruction and data TLBs. The experimental setup involved multiple SoftSDV sessions connected over virtual network. We choose to run iperf session for the sake of brevity to study life of I/O packet. The iperf client is executed to initiate packet transmissions from a Xen environment.

4.1 Life Cycle of an I/O packet

We describe the execution flow of packet processing inside a Xen virtual machine. In Figure 6, we present an overview of different stages which characterize the life of a packet between VM domains. Typically, a network packet in the Xen environment goes through the following four stages in its life cycle:

1. Unprivileged domain – packet build and memory allocation
2. Page transfer Mechanism – a zero-copy mechanism to map pages in virtual address space of Dom0/DomU domains
3. Context Switch between hypervisor and domains – timer interrupts
4. Privileged domain – Event channel mechanism to send acknowledgment to guest domain.

EIP	Module	Function Name	Instr Count
c02f1910	Dom-U	tcp_transmit_skb	0
c02b37a0	Dom-U	skb_clone	25
c02f16f0	Dom-U	tcp_select_window	238
c02f30e0	Dom-U	__tcp_select_window	250
c02f1780	Dom-U	tcp_build_and_update_options	362
c02f1550	Dom-U	tcp_event_data_sent	454
c0331a90	Dom-U	_read_lock_bh	702
c0331c60	Dom-U	_read_unlock_bh	728
c0331a90	Dom-U	_read_lock_bh	800
c0331c60	Dom-U	_read_unlock_bh	885
c0331b40	Dom-U	_spin_lock	954
c02cbaf0	Dom-U	qdisc_restart	1000
c0331920	Dom-U	_spin_trylock	1051
c03319d0	Dom-U	_spin_lock_irq	1097
c023f6a0	Dom-U	gnttab_claim_grant_reference	1129
c023f360	Dom-U	gnttab_grant_foreign_access_ref	1159
c023e500	Dom-U	notify_remote_via_irq	1211
ff127900	Hypervisor	copy_from_user	1262
ff127860	Hypervisor	__copy_from_user_ll	1281
ff105130	Hypervisor	evtchn_send	1330
ff104d80	Hypervisor	evtchn_set_pending	1370
ff10e780	Hypervisor	vcpu_wake	1408

Figure 8: TCP transmit and Grant table invocation (transition from period (a) to period (b) in Figure 6).

4.1.1 Unprivileged Domain

On the transmit side, a packet originates from the iperf application. The execution flow traverses from the application into the DomU guest OS kernel where all the required TCP/IP processing is completed. The TCP/IP stack builds the payload in transmit socket buffers (skb) and hands them over to the front-end driver. Socket buffers (skb) represent network packets as they are transmitted through the system and facilitate the implementation of zero-copy networking between Xen virtual machines [32]. For example, we identified an interface in Xen to allocate a socket buffer in the networking layer (alloc_skb_from_cache). The front end driver uses the grant table mechanism provided by the hypervisor to hand over the buffer to Dom-0. Figure 7 shows the functions and the associated instruction count for overall life of the packet in DomU: lock socket, copy data from user space to kernel space, allocate page from free list, and release socket lock. Note that the instruction count statistics are shown in chronological order with function entry points as markers, it is not done at individual function level. We removed some repeating functions to improve readability. As part of the transmit processing the DomU guest domain communicates to Dom0 using event channels.

4.1.2 Grant Table Mechanism

Once the message to notify Dom0 of a transmit request is sent through event channels, the transmit packets are picked up by the Dom0 when the hypervisor schedules it to execute. The Xen VMM provides a generic mechanism to share memory pages between domains, referred to as grant table mechanism: before sending an event to Dom0, the DomU guest domain sets access rights to the memory pages holding the actual packet contents through a grant table interface provided by the hypervisor. Figure 8 demonstrates the execution flow from domU to hypervisor through the grant table mechanism.

4.1.3 Timer Interrupts

Switching into the hypervisor is initiated typically on timer interrupts. The functions invoked during a timer interrupt which results in a VM switch are shown in Figure 9. The last function in the table is invoked inside the Dom0.

EIP	Module	Function	Instr count
ff114210	Hypervisor	smp_apic_timer_interrupt	0
ff110760	Hypervisor	stop_timer	94
ff1103c0	Hypervisor	remove_entry	119
ff110650	Hypervisor	set_timer	485
ff110540	Hypervisor	add_entry	517
ff110370	Hypervisor	up_heap	539
ff124570	Hypervisor	update_dom_time	679
ff117820	Hypervisor	context_switch	723
ff117690	Hypervisor	__context_switch	752
ff11d820	Hypervisor	write_ptbase	856
ff127810	Hypervisor	__copy_to_user_ll	931
ff10f980	Hypervisor	do_softirq	991
ff124280	Hypervisor	get_s_time	1026
ff114120	Hypervisor	reprogram_timer	1105
ff124280	Hypervisor	get_s_time	1116
ff1493d5	Hypervisor	create_bounce_frame	1260
c023da30	Dom-0	evtchn_do_upcall	1350

Figure 9: Context switch between hypervisor and Dom-0 VM - Timer interrupts (transition from period (b) to (c) in Figure 6).

4.1.4 Privileged Domain

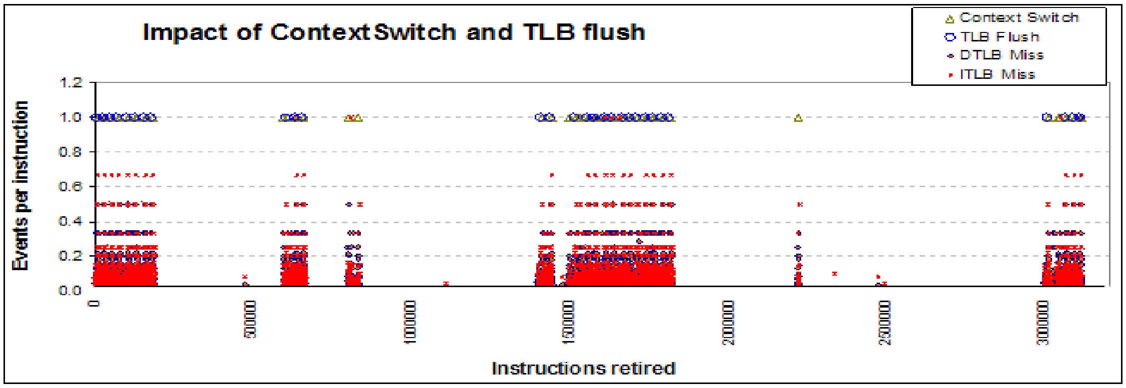
Once inside the Dom0, the backend driver picks up the packets and bridges them to the real network interface card. For this it needs to access the packet buffer from the guest domain. It uses the grant provided by the guest to map the page into its own domain and accesses it. Once transmit processing is complete, Dom0 sends an acknowledgment back to the DomU guest domain using event channel mechanisms. Execution flow in Dom0 is shown in Figure 10 (since the complete execution at this stage long, we only are able to show snippets of execution covering the basic flow and highlighting the important functions). Note that the grant table mechanism is used to map guest pages into Dom0 address domain on the backend receiving side. Then the packet is sent to the bridge code, after which it is sent out on the wire. Once complete, the host map is destroyed and an event is sent on the event channel to the guest domain.

It is interesting to note that the processor TLB is flushed while destroying the grant. It is done by writing the CR3 register (the x86 page table pointer) through the *write_cr3* function. We look at the impact of this TLB flush in Section 4.2. This completes the transmit side processing.

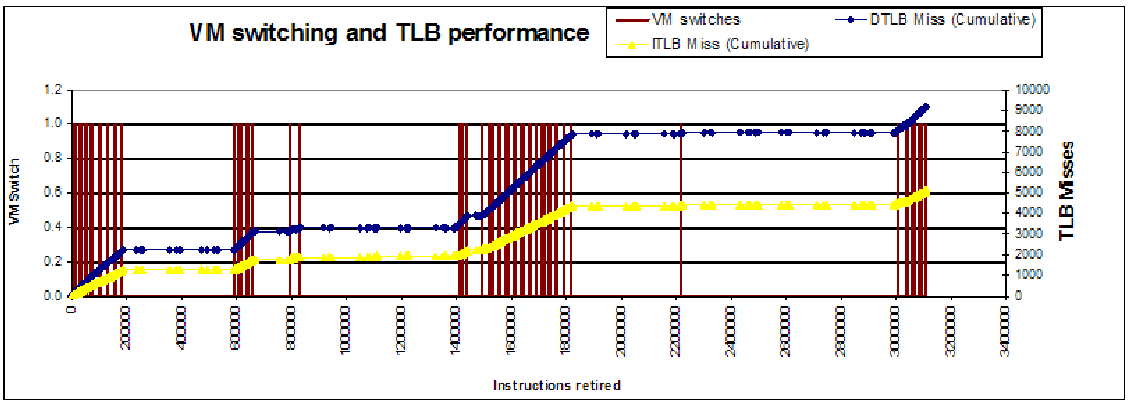
EIP	Function Name	Module	Instr.Count
c023da30	evtchn_do_upcall	Dom0	0
c0142d30	__do_IRQ	Dom0	85
c0331b40	_spin_lock	Dom0	116
c023e530	mask_evtchn	Dom0	147
c0142c80	handle_IRQ_event	Dom0	193
c0249700	add_to_net_schedule_list_tail	Dom0	243
ff11e8b0	cleanup_writable_pagetable	Hypervisor	1577
ff103ac0	find_domain_by_id	Hypervisor	1741
ff1205f0	create_grant_host_mapping	Hypervisor	1963
ff11dcd0	put_page_from_l1e	Hypervisor	2033
ff127810	__copy_to_user_ll	Hypervisor	2143
c02c9cb0	eth_type_trans	Dom0	2550
c02b98c0	netif_rx	Dom0	2671
c02b9de0	netif_receive_skb	Dom0	3170
c02d4f40	nf_hook_slow	Dom0	3412
e121ffb0	br_handle_frame_finish	Dom0	4258
e121ed70	br_fdb_update	Dom0	4294
e121e7d0	__br_fdb_get	Dom0	4475
e121f2d0	br_forward	Dom0	4630
c02d4f40	nf_hook_slow	Dom0	4686
e1031ad0	e100_tx_clean	Dom0	18899
c0331b40	_spin_lock	Dom0	18914
e1030000	e100_enable_irq	Dom0	19004
c0331990	_spin_lock_irqsave	Dom0	19015
c0331b90	_spin_unlock_irqrestore	Dom0	19043
ff11e8b0	cleanup_writable_pagetable	Hypervisor	19545
ff127860	__copy_from_user_ll	Hypervisor	19625
ff103ac0	find_domain_by_id	Hypervisor	19727
ff1203b0	**destroy_grant_host_mapping**	Hypervisor	19827
ff127810	__copy_to_user_ll	Hypervisor	19972
ff123b40	**__flush_tlb_mask**	Hypervisor	20054
ff119970	**write_cr3**	Hypervisor	20072
c024a0f0	make_tx_response	Hypervisor	20266
ff127900	copy_from_user	Hypervisor	20422
ff127860	__copy_from_user_ll	Hypervisor	20451
ff105130	**evtchn_send**	Hypervisor	20531
ff104d80	evtchn_set_pending	Hypervisor	20589
c01343c0	rcu_pending	Dom0	21106

Figure 10: Life of a packet in Dom-0: Accessing granted page, ethernet transmission, destroy grant mapping and event notification back to hypervisor (period (c) in Figure 6). Dotted line indicates unidentified function calls.

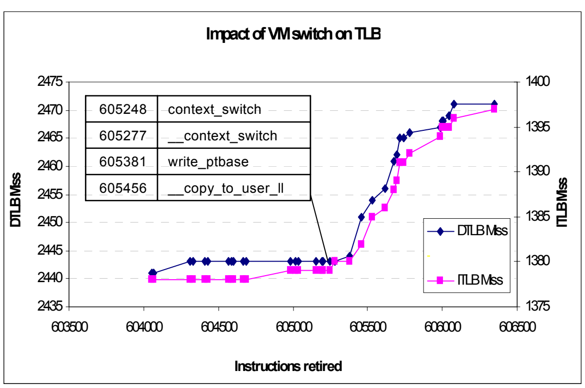
Note that the flow described here is only an example. The execution flow may vary based on the state of the stack and the availability of buffers. External interrupts also may alter the execution flow considerably. An execution driven simulation environment allows us to profile various execution flows and characterize the I/O architecture correctly. Similarly, we can get the execution flow at the receiver side in a Xen execution environment



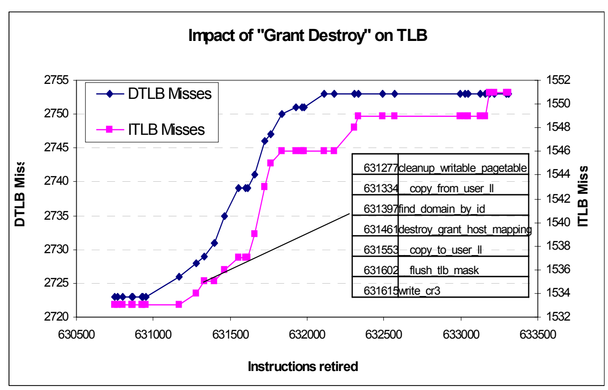
(a)



(b)



(c)



(d)

Figure 11: (a) Execution path showing the impact of TLB flush can context switch on TLB misses (b) Correlation between VM switching and TLB misses (c) TLB misses following a VM Switch (d) TLB misses following a grant destroy**.**

4.1.5 Cache and TLB characteristics

It is important to analyze the impact of hardware design decisions on the performance of VMMs. As mentioned earlier, we focus on the performance characteristics related to cache and TLB resources. Figure 11(a) shows an execution snippet where TLB flushes and misses are plotted as a function of simulated instructions retired. The figure shows that there is a high correlation between the TLB misses, context switches and TLB flush events. An execution run of VM during a period with no context switches or TLB flushes results in negligible TLB misses.

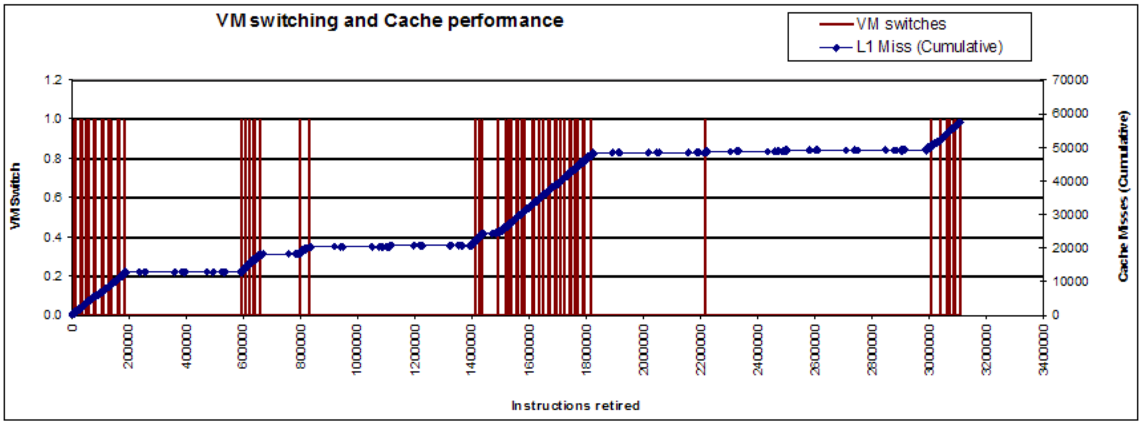


Figure 12: Execution path showing the impact of VM switch on cache misses.

Whenever TLB flushing events happen, there is a surge of TLB misses. This correlates well with the observations of TLB miss overhead in earlier studies. Figure 11(b) shows the increased number of TLB misses associated with the VM switches in a cumulative graph. We observe that there is a surge of TLB misses associated with each VM switch. Execution segments without VM switches show flat areas with few TLB flushes.

Figure 11(c) depicts a typical VM switch scenario. The execution moves from one VM to another through a context. The CR3 value is changed to point the new VM context. This triggers the hardware to flush all the TLBs to avoid invalid translations. But this comes with a cost of TLB flushes every time a new page is touched, both for code and data pages.

Another scenario is the explicit TLB flushes done by the Xen hypervisor as part of the data transfer between VMs. This is an artifact of the current IO VM implementation as explained in the previous section. In order to revoke a grant, a complete TLB flush is executed explicitly, which also creates TLB performance issues similar to VM switch. Figure 11(d) demonstrates the code flow and the TLB impact.

Figure 12 shows the impact of context switches on cache performance. The vertical lines mark VM switch events obtained through symbol annotation, and the plotted line shows the cumulative cache miss events. Note that the cache miss rate increases are also correlated with VM switch events.

4.2 Cache and TLB Scaling

In this section, we look at the impact of cache size and TLB sizes on I/O virtualization overhead. As described earlier, we used the functional model of SoftSDV to boot a RHEL 4 Linux disk image and Xen-3.0.2 as a test bed. We ran two sessions of the SoftSDV simulation tool connected to each other through a virtual subnet configured for network communication. For each experiment, we executed a session of iperf [9]; TLB and cache statistics we measured for transfer of approximately 25 million TCP/IP packets.

Figure 13 shows the cache scaling effect. We simulated a two level cache: 32KB L1 (split data and instruction) and a 2MB unified L2 cache. The primary goal is to understand the cache sensitivity of the I/O virtualization architecture in the context of network I/O. Note that increasing the L2 cache size up to 4MB provided good performance scaling, after which the increase in performance was minimal. Increasing the cache size beyond 8MB, the miss rate the rate of reduction in miss rates is small. We can attribute reduced miss rates from the 8MB cache to the inclusion of needed pages from hypervisor, Dom0 and DomU.

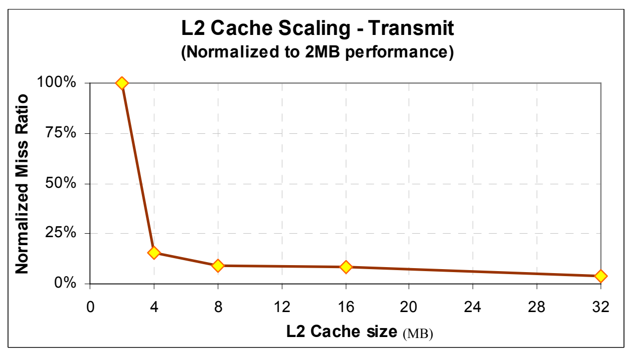


Figure 13: Transmit L2 Cache Scaling

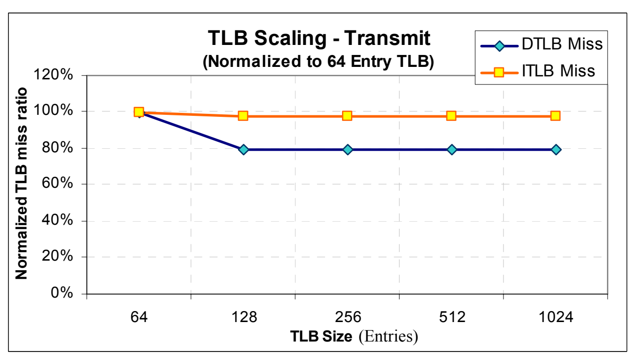


Figure 14: Transmit TLB Scaling Impact

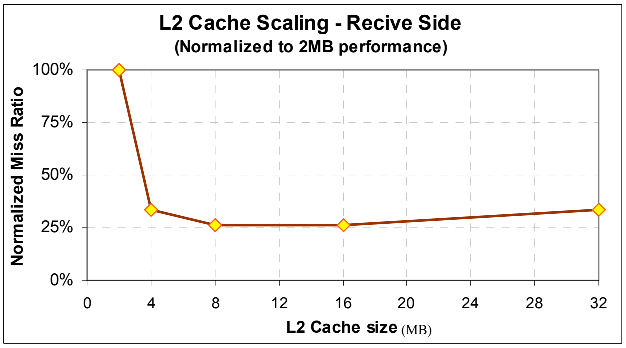


Figure 15: Receive L2 Cache scaling

Figure 14 shows the TLB performance scaling impact for data and instruction TLBs. As shown in the figure, with increase in size of the data TLB, the miss ratio decreases for sizes up to 128 entries. For larger sizes, the miss ratio is nearly constant. The ITLB miss rate decreases slightly, while the DTLB rate shows a sharper decrease from 64 to 128 entries. We infer that TLB size of 128 entries is sufficient to incorporate all address translations during the TLB stage. Similarly, we also performed the cache and TLB scaling studies on the receive side. Results are given in Figures 15 and 16 respectively.

Finally, we studied the potential impact of a TLB optimization to make global hypervisor pages persistent in TLBs. In the absence of TLB tagging, on a TLB flush all translations are invalidated. The goal of this optimization is to allow tagging the TLB with a single bit indicating that tagged translations are not to be flushed, which can be used in a virtualized environment to tag pages associated with hypervisor code and data. As shown in Figure 17, such an optimization indeed has the potential to substantially reduce DTLB misses (and, to a lesser extend, reduce ITLB misses).

5. Related Work

The characterization of the performance overhead is an important concern in the study of virtualized environments, and several studies have addressed this issue with a methodology based on execution of application benchmarks on virtualized platforms [15, 20]. Performance monitoring tools have been deployed to gauge application performance in virtualized environments [3, 4, 5]. Traditional network optimizations such as TCP/IP checksum offload, TCP segmentation offload are being used to improve network performance of Xen-based virtual machines [4]. In addition, faster I/O channel for transferring network packets between guest and driver domains is being studied [4]. These studies lack micro-architectural overhead analysis of the virtualized environment.

TLB misses after context switches negatively impact I/O performance. In the past, TLBs have been tagged with a global bit to prevent flushing of global pages such as shared libraries and kernel data structures. In current system architectures, context switch overhead can be reduced through tagging TLB entries with address-space identifiers (ASID). A tag based on VMID could be further used to improve I/O performance for virtual machines. Processor architectures, with hardware virtualization support, incorporate features such as virtual-processor identifiers (VPID) to tag translations in the TLB [6, 19].

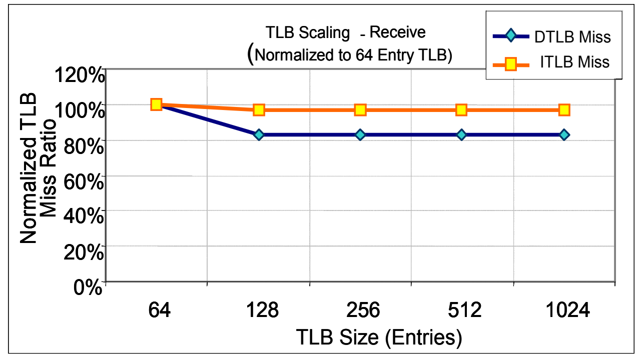


Figure 16: Receive TLB Scaling

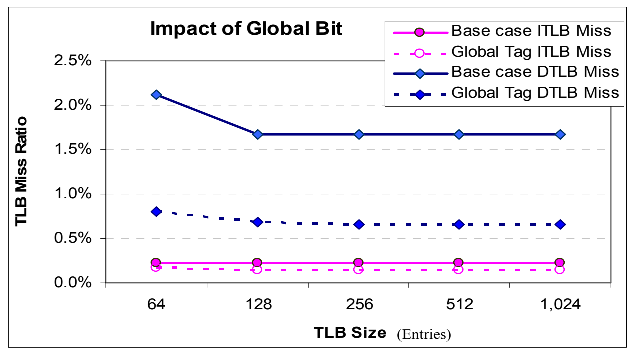


Figure 17: Impact of Global bit for transmission

6. Conclusion and Future Work

The focus of this paper is to present a case study of a virtualized workload in a simulated environment to study micro-architectural features as a means of performance evaluation. We used an execution driven simulation framework, along with a symbol annotation methodology, to analyze the overheads of an I/O intensive workload running in a virtualized Xen environment. We also presented the initial research results from TLB and cache scaling for the I/O workload. The execution driven simulation framework presented in this paper provides the speed and flexibility needed for understanding the current architecture bottlenecks and experiment with potential architectural changes in hardware and software.

We plan to extend the studies with VPID tagged TLBs and also understand the impact of hardware based TLB coherence management. We will also be investigating the feasibility of hardware support for better inter-VM communication mechanisms using an extended analysis framework. The importance of performance isolation and VM level QoS [7, 25] is a growing research area especially with the introduction of multi-core processors sharing platform resources like cache, TLB and memory resources. We are investigating hardware and software enhancements for architecting QoS aware multi-core platforms.

References

[1] A. Foong, J. Fung, and D. Newell, “An In-Depth Analysis of the Impact of Processor Affinity on Network Performance”, Proceeding of IEEE Int’l Conf. Networks, IEEE Press, 2004.

[2] A. Foong, T. R. huff, H. H. Hum, J. P. Patwardhan, and G. J. Regneir, “TCP Performance Analysis Re-visited,” International Symposium on Performance Analysis of Software and Systems, Mar 2003.

[3] A. Menon, J. R. Santos, Y Turner, and G. Janakiraman, “Xenoprof - Performance profiling in Xen”. http://xenoprof.sourceforge.net/xenoprof_2.0.txt (Accessed June 2006).

[4] A. Menon, A. Cox, W. Zwaenepoel, “Optimizing Network Virtualization in Xen”, 2006 USENIX Annual Technical Conference.

[5] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman and W. Zwaenepoel, “Diagnosing Performance: Overheads in the Xen Virtual Machine Environment”, 1st ACM/USENIX Conference on Virtual Execution Environments (VEE’05), June 2005.

[6] G. Neiger, A. Santoni, F. Leung, D. Rodgers and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. Intel Technology Journal, August 2006.

[7] H. Kannan, F. Guo, L. Zhao, R. Illikkal, Ravi Iyer, D. Newell, Y. Solihin and C. Kozyrakis, “From Chaos to QoS: Case Studies in CMP Resource Management,” 2nd Workshop on Design, Architecture and Simulation of CMP platforms (dasCMP / MICRO), Dec 2006.

[8] Intel Virtualization. Technology Specification for the IA-32 Intel Architecture, April 2005.

[9] Iperfbenchmark. http://dast.nlanr.net/Projects/Iperf/ (Accessed June 2006).

[10] J. E. Smith, and R. Nair. Virtual Machines: versatile platforms for systems and processes. Morgan Kaufmann publishers. May 2005.

[11] J. F. Kloster, J. Kristensen and A. Mejlholm, ”Efficient Memory Sharing in the Xen Virtual Machine Monitor”, Technical Report. Department of Computer Science, Aalborg University, Jan 2006.

[12] J. Liu, W. Huang, B. Abali and DK Panda, “High Performance VMM-Bypass I/O in Virtual Machines”, USENIX Annual Technical Conference, June 2006.

[13] J. Sugerman, G. Venkitachalam, and B.Lim, “Virtualizing I/O devices on VMware Workstation’s hosted virtual machine monitor”, USENIX Annual Technical Conference, June 2001.

[14] L Albertsson and Peter S Magnusson, “Simulation-based temporal debugging of Linux”, Proceeding of 2nd real-time linux workshop 2000.

[15] L. Cherkasova and R. Gardner, ”Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor,” Proceedings of the USENIX Annual Technical Conference, April 2005.

[16] M. Ramachandran, N. Smith, M. Wood, S. Garg, J. Stanley, E Eduri, R. Rappoport, A. Chobotaro, C. Koltz and L. Janz, “New Client Virtualization Usage Models Using Intel Virtualization Technology”, Intel Technology Journal, August 2006.

[17] M. Rosemblum and T. Garfinkel, “Virtual Machine Monitors: Current Technology and Future trends”, IEEE Computer, 38(5): 39-47, 2005.

[18] M. T. Yournst. PTLsim/X: Xen and the Art of full system multiprocessor simulation. http://www.ptlsim.org/Documentation/PTLsimManual.pdf (Accessed Dec 2006).

[19] “Pacifica – Next Generation Architecture for Efficient Virtual Machines”. http://developer.amd.com/assets/WinHEC 2005_Pacifica_Virtualization.pdf (Accessed April 2007).

[20] 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”, Proceedings of the ACM symposium on operating systems principles, Oct 2003.

[21] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, B. Werner, “Simics: A Full System Simulation Platform”, IEEE Computer, Feb 2002.

[22] R. Figueiredo, P. Dinda, J. Fortes, “Resource Virtualization Renaissance”, guest editor’s introduction, IEEE Computer, 38(5): 28-31, 2005.

[23] R. Illikkal, R. Iyer, Don Newell, “Micro-Architectural Anatomy of a Commercial TCP/IP Stack”, 7th IEEE Annual Workshop on Workload Characterization (WWC-7), Oct. 2004.

[24] R. Iyer, “On Modeling and Analyzing Cache Hierarchies using CASPER”, 11th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'03).

[25] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu and S. Reinhardt, “QoS Policies and Architecture for Cache/Memory in CMP Platforms,” accepted for ACM SIGMETRICS 2007.

[26] R. Uhlig, R. Fishtein, O. Gershon, I Hirsh and H. Wang, “SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture. Intel Technology Journal. Q4, 1999. (http://www.intel.com/ technology/itjf).

[27] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer, “Are Virtual Machine Monitors Microkernels Done Right?”, Proceedings of Hot OS 2005.

[28] SimOS: The Complete Machine Simulator. http://simos.stanford.edu/ (Accessed Dec 2006)

[29] SimpleScalar: Tools for simulation of modern processors http://www.cs.wisc.edu/~mscalar/software.html (Accessed Dec 2006).

[30] S. Makineni and R. Iyer, “Performance Characterization of TCP/IP Packet Processing in Commercial Server Workloads”, 6th IEEE Workshop on Workload Characterization, Oct 2003.

[31] V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser, "Performance of address space multiplexing on the Pentium," Tech. Rep. 2002-1, University of Karlsruhe, Germany, 2002.

[32] Xen networking. http://lwn.net/Articles/117736/ (Accessed September 2006).