Ally: OS-Transparent Packet Inspection Using Sequestered Cores

Jen-Cheng Huang[*] Matteo Monchiero[†] Yoshio Turner[‡] Hsien-Hsin S. Lee[*]

[*]Georgia Institute of Technology [†]Intel Labs [‡]HP Labs
tommy24,leehs@gatech.edu matteo.monchiero@intel.com yoshio.turner@hp.com

ABSTRACT

This paper presents Ally, a server platform architecture that supports compute-intensive management services on multicore processors. Ally introduces simple hardware mechanisms to sequester cores to run a separate software environment dedicated to management tasks, including packet processing software appliances (e.g. for Deep Packet Inspection, DPI) with efficient mechanisms to safely and transparently intercept network packets. Ally enables distributed deployment of compute-intensive management services throughout a datacenter. Importantly, it uniquely allows these services to be deployed independent of the arbitrary OSs and/or hypervisor that users may choose to run on the remaining cores, with hardware isolation preventing the host environment from tampering with the management environment. Experiments using full system emulation and a Linux-based prototype validate Ally functionality and demonstrate low overhead packet interception; e.g., using Ally to host the well-known Snort packet inspection software incurs less overhead than deploying Snort as a Xen virtual machine appliance, resulting in up to 2x improvement in throughput for some workloads.

1. INTRODUCTION

Packet processing services like Deep Packet Inspection (DPI) are used in datacenters for functions including intrusion detection, content insertion, performance monitoring, traffic classification, and flow management [1]. These services are provided by specialized appliances deployed at the boundary (e.g. wide-area network gateway) between the datacenter and the external network to process traffic entering or exiting the datacenter.

This approach is poorly suited to processing traffic that remains local to the datacenter, even though local traffic is growing in importance. To exploit economies of scale, modern datacenters consolidate many interconnected applications and services. This trend is clear for both private Enterprise datacenters and public cloud computing datacenters, which support a fast-changing multitude of mutually untrusted tenants. These models call for packet processing services to be flexibly placed on-demand throughout a datacenter rather than just at the external boundary. An attractive approach would co-locate packet processing with user applications throughout a datacenter [2], enabling packet processing services to leverage the abundant compute and memory resources on commodity servers.

This paper presents Ally, a server architecture that provides the basic building block for distributed packet processing services. Ally adds hardware partitioning to a server, enabling it to run two parallel and independent software stacks – one stack for user applications and OS or hypervisor, and a second software stack for packet processing. Ally also adds processor support for OS-transparent network packet interception by the packet processing stack. Finally, Ally reuses the existing management network interface on the server for datacenter administrators to deploy and control packet processing services.

As we enter the era of processors with a large number of inexpensive cores, the approach of Ally is to partition a processor into a "privileged" set of cores that execute packet processing services, and an "unprivileged" set of cores that execute user applications and operating systems. Completely separate software stacks can run in parallel in the two partitions. Ally's hardware extensions are intentionally minimal. They are invoked only on low frequency I/O actions (interrupts and memory-mapped I/O accesses), with no impact on other computation and negligible impact on processor clock speed and power consumption.

Using multicore partitioning for packet processing has important advantages over the alternative approaches of enhancing NICs or deploying packet processing in hypervisor-hosted virtual machines (VMs). Compared to enhancing the NIC, using CPU cores allows new functionality to be provided in software without hardware changes, and can leverage the large capacity of server memory and the high performance of modern processors. General-purpose cores may also be used in future processors to control on-chip hardware accelerators like GPUs or regular expression engines. Moreover, a CPU-based approach is not limited to packet processing

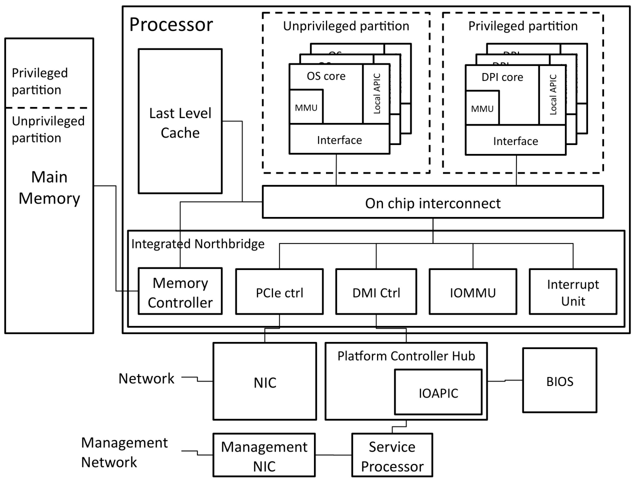


Figure 1: Target platform architecture

but could also support management services related to storage, power, etc. Compared to using VMs for isolation, Ally's hardware partitioning may be more reliable by avoiding resource and fate sharing, and by presenting a narrower attack surface than the full set of hypercall APIs. Hardware partitioning also preserves existing software generality instead of forcing datacenters to use a common hypervisor on every server. Ally preserves and extends existing server management interfaces and capabilities and supports both virtualized and non-virtualized OSs.

We carried out functional validation of Ally hardware extensions using the QEMU [3] full system emulator. To evaluate Ally's performance, we built a software prototype that modifies the Linux kernel to emulate Ally hardware functions. Using the well-known Snort packet inspection software, our experimental results show that Ally+Snort has acceptably low overhead for packet interception. For the workloads we studied, Ally achieves from 56% to 100% higher throughput than a system that uses a Xen hypervisor driver domain to transparently intercept and inspect packets.

The rest of the paper is organized as follows. Section 2 describes our assumptions about the platform architecture and the NIC-OS model. Section 3 describes the trust and threat model motivating the design of Ally. Section 4 describes the Ally architecture, including core sequestering, memory protection, and packet interception mechanisms. Section 5 presents the experimental evaluation. Section 6 discusses related work, and Section 7 concludes.

2. ASSUMPTIONS

This section describes our baseline architecture and reviews the interaction between an OS and a NIC.

2.1 Platform Architecture

Figure 1 is a simplified block diagram of a generic platform architecture, built around a multicore processor with many cores sharing a Last Level Cache (LLC). Cores are logically grouped into a privileged partition and an unprivileged partition. Privileged cores run management applications, while unprivileged cores run OS/hypervisor and user applications. As we focus on DPI as a primary use-case, we often refer to the cores in the privileged partition as *DPI cores* and the cores in the unprivileged partition as *OS cores*.

Each core has a Memory Management Unit (MMU) for virtual-to-physical address translation, and a local interrupt controller (Local APIC). An integrated Northbridge has a Memory (DRAM) Controller, PCIe I/O controller, Interrupt Unit to route interrupt signals to the proper Local APIC from the IOAPIC or through Message Signaled Interupts (MSI), I/O Memory Management Unit (IOMMU), and a point-to-point high speed link (e.g., Intel DMI) connecting to a Platform Controller Hub (PCH).

A service processor (for example Intel iAMT [4] or HP iLO [5]) is used to manage the platform. The service processor interfaces to the CPU via the PCH. The service processor has a a dedicated network interface typically attached to a dedicated management network. With Ally, datacenter administrators use this management network to deploy packet processing services in the privileged partition.

2.2 OS-NIC Interaction

We describe OS-NIC interaction using the popular Intel Pro/1000 NIC. The same description applies to most modern NICs with only minor differences.

The NIC has transmit/receive buffers and a set of device registers mapped into the system memory address space. One register points to a transmit descriptor queue and another one to a receive descriptor queue. Both queues reside in the host main memory. Each descriptor contains a pointer to packet data that must be transmitted or to receive buffers. For each queue, memory-mapped Head Pointer and Tail Pointer registers record the head and tail position in the queue as buffers are posted and as packets are received and transmitted.

The OS advances the Tail Pointer to notify the NIC that some descriptors have been posted and await being consumed (either transmitted or received). The NIC uses Direct Memory Access (DMA) to transfer descriptors and packets between the NIC buffers and the main memory. Once a transfer has completed, the NIC updates the Head Pointer and raises a completion interrupt. Most NICs use interrupt coalesce timers to reduce interrupt frequency and associated software processing overheads by waiting for a batch of packet I/Os to be completed before signaling an interrupt.

The OS may determine which I/Os have completed by reading the Head Pointer or by inspecting the status of the queued descriptors.

3. TRUST AND THREAT MODEL

The trusted entities of an Ally-based system are the hardware, the firmware (BIOS), and the software running on the privileged (DPI) cores. We assume that the BIOS FLASH chips include hardware protection against re-writing by OSs. For maximum security, the firmware and the privileged cores' software are not directly exposed to the external network. Instead, a dedicated management channel is provided to configure the BIOS and to deploy services on the DPI cores, including platform-specific device drivers. This channel uses a distinct network interface connected to a management network via a service processor, as shown in Figure 1. Additionally, the DPI cores can use this channel to raise management alerts when suspicious traffic is detected. Each hardware extension in Ally is controlled through memory-mapped I/O (MMIO) configuration registers accessible only to the privileged cores and exposed to the datacenter via the management channel.

Ally can be used to deploy a DPI engine to protect against transmission and reception of malicious network traffic for the workloads running on unprivileged cores. As described in Section 4.3, Ally offers configurable mechanisms for packet interception. This allows administrators to select the degree of protection provided depending on the perceived threat level of the workloads running in the unprivileged domain, allowing a configurable trade-off between protection and cost/performance.

Denial of service attacks where software on unprivileged cores guesses the Local APIC ID of the DPI-Core and generates corresponding interrupts can be partially avoided by having the DPI cores configure their Interrupt Descriptor Tables to ignore all interrupts from non-authorized devices. However, the motivation for such attacks seems low because they would only hurt the attacker's own I/O performance.

4. ARCHITECTURE

Ally provides an ensemble of hardware, firmware, and software mechanisms. Ally creates two partitions: a privileged partition for DPI cores, and an unprivileged partition for OS cores. Physical memory is split in two separate regions, one for each partition, and independent software stacks boot in each partition. Memory protection is extended to prevent OS cores from reading or writing the memory region of the DPI cores, while allowing DPI cores in the privileged partition to access OS memory. Simple mechanisms enable DPI cores to intercept packets exchanged between OS cores and NIC.

Table 1: Hardware/Firmware Modifications

Unit	Modification
OS cores' MMU	Prevent memory accesses to DPI memory from OS cores
IOMMU	Prevent non authorized DMA to DPI memory
IDT	Prevent non authorized interrupts from OS cores to DPI cores
BIOS	Boot DPI appliance and hide DPI cores/ memory
Interrupt Unit	Redirect NIC interrupts to DPI cores
OS cores' MMU	Redirect MMIO register accesses to DPI memory
All units	Extra MMIO registers to configure Ally functionalities

Table 1 summarizes the hardware modifications.

4.1 Core Sequestration

Core sequestration enables distinct software environments to be booted in each partition. Conventional Intel multiprocessor (MP-compliant) systems use a standard boot procedure at startup. Ally works with this boot procedure to launch the DPI environment on the DPI cores and to conceal the DPI cores from the OS cores.

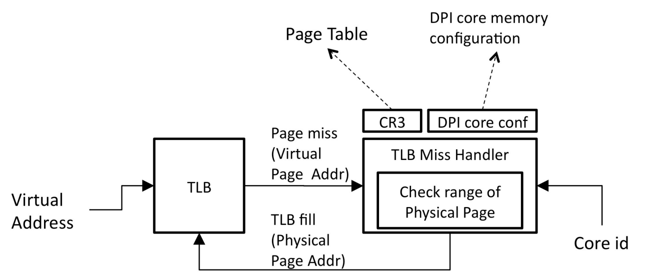
In the conventional procedure, a core (called BSP core) wakes up the other cores (called AP cores). Each AP core runs code loaded from BIOS which executes a self-test and enters the core's unique identifier (Local APIC ID) into the ACPI's Multiple APIC Description Table (MADT) and the MP configuration table. The OS uses the MADT and the MP configuration table to find all information needed to discover and communicate with a core. After initialization, each AP core halts and waits for an Inter-Processor Interrupt (IPI) to resume execution.

Ally uses a modified BIOS that selects some AP cores as DPI cores and loads a custom initialization procedure onto each DPI core. The custom procedure deletes the DPI core's entry from the MADT and the MP configuration table, thus hiding the DPI core from the OS cores. The DPI cores then load the custom Interrupt Descriptor Table and begin to load the DPI software environment.

The DPI cores interact with the built-in service processor to request the most up-to-date code from a remote management server. The service processor downloads an executable image via the management network into DPI memory for the DPI cores to execute.

4.2 Memory Protection

Ally splits the physical address space in two regions identified by a simple range check, i.e., checking whether physical addresses are above or below a dividing boundary line. Ally adds a memory-mapped I/O (MMIO)



Page Table
DPI core memory configuration
CR3
DPI core conf
TLB Miss Handler
Check range of Physical Page
TLB
Page miss (Virtual Page Addr)
TLB fill (Physical Page Addr)
Virtual Address
Core id

Figure 2: Modified Memory Management Unit. Ally uses a modified TLB Miss Handler to protect from unauthorized memory accesses

configuration register to the MMU for each core that stores the boundary address between DPI and OS memory. This range check is much simpler than adding a new level of address translation, and is fully compatible with the existing address translation hierarchy, including nested or extended page tables.

To prevent OS cores from accessing DPI memory, the MMU needs a simple address check on each hardware page table walk, to verify that the translated physical address falls in an accessible region for the core that caused the access. The modified TLB Miss Handler (TMH), shown in Figure 2, verifies that any physical address, which is being loaded in the TLB on an OS core, is not in the range of DPI memory. This address check has very low overhead on the performance of the system. It has no overhead on TLB hits, since it happens only for TLB misses. In this case, the address comparison is likely to take a fraction of cycle, a negligible overhead for a TLB miss, which may have orders of magnitude higher latency.

Ally uses similar checks for the I/O Memory Management Unit (IOMMU) to prevent non-authorized devices from performing DMA writes to DPI memory, and to protect DPI memory in x86 real address mode (usually only needed early in the boot process).

4.3 Packet Interception

Ally enables DPI cores to transparently intercept, examine and potentially modify packets in both transmit and receive directions between OS and NIC. Packet interception is accomplished by virtualizing the NIC descriptor queues. Ally maintains a copy of each descriptor queue in DPI memory and configures the NIC to use these queues instead of the actual driver queues. The DPI cores are thus responsible for synchronizing the DPI queues with the OS queues.

DPI cores inspect packets referenced by the enqueued descriptors. Once packet processing is done, DPI cores transfer descriptors between the virtual descriptor queues seen by the device driver on OS cores and the real NIC descriptor queues in DPI memory.

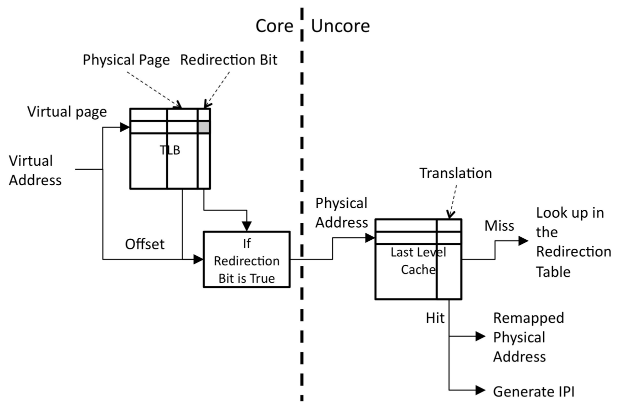


Figure 3: Address redirection flow, involving TLB redirection and access to the Redirection Table cached in the Last Level Cache

The DPI cores interpose between the OS and the NIC. This interposition is logically transparent to both the OS (or hypervisor) running on the OS core and the NIC because the usage of the descriptor queue is device dependent, not OS dependent. Thus, no modification is needed to the OS (including the device driver) or the NIC.

The DPI cores download modified NIC drivers via the service processor. The modifications are mostly on the transmit and receive paths, which previous work has shown comprises only a small portion of the driver code, and which can even be separated out using an automated process [6]. Modified drivers would likely be provided by the server vendor to increase the value of their server management components. Development costs would be reasonable, since each vendor of datacenter-class servers ships only a small set of NIC models to enjoy large volume cost savings and simplify certification.

Since an OS interacts with a NIC through interrupts and MMIO operations, Ally provides configurable redirection of these operations to DPI cores for I/O interception. A DPI core specifies which NIC interrupts (IOAPIC and MSI) are to be redirected, and the modified Interrupt Unit in the Northbridge steers the selected interrupts to DPI cores. DPI core software uses Inter-Processor Interrupts (IPIs) to mimic NIC interrupts back to the OS.

Any MMIO access performed by an OS core and directed to the NIC descriptor queue registers must be redirected to a reserved area of DPI memory. This leaves the DPI cores in control of the NIC real registers. The accesses that need to be redirected include accesses to the base pointers of the descriptor queues and the Tail/Head registers of each queue. In addition, accesses to interrupt-related registers must also be redirected. This allows the DPI cores to control the interrupt rate based on the packet processing speed and allows the OS cores to transparently use techniques like Linux NAPI.

Ally enhances the MMU and TLB to provide MMIO access redirection. Figure 3 shows an efficient implementation. All MMIO remappings are specified in a Remapping Table in DPI memory. This table is small, requiring less than ten entries per intercepted I/O device. Virtual address translation is extended to check the Remapping Table on OS core memory accesses. The TLB Miss Handler (TMH) performs the usual page table walk. If the resulting physical address refers to an uncacheable page (hence potentially an MMIO page), then the TMH checks the Redirection Table to see if any address in the table belongs to the faulting page. If so, the TMH sets a new "redirection" bit in the TLB entry.

OS cores accesses, that hit in the TLB and have the redirection bit set, also trigger a Remapping Table lookup to determine if the address within the page has a remapping. If a remapping is found, the MMU raises an IPI to notify a DPI core of the access attempt by the OS core.

For normal memory accesses, the redirection bit is not set, incurring no extra overhead. For the minority of accesses that need Remapping Table lookup, Ally uses the Last Level Cache (LLC) to cache the small Remapping Table, speeding up table lookup. Overall, Remapping Table lookup slightly increases the latency of MMIO accesses while preserving the speed of normal memory accesses and non-redirected MMIO accesses.

We next describe the steps taken by DPI cores to intercept packets and virtualize the device queues. Since the real Head/Tail Pointer registers in the NIC refer to the queues in DPI memory (*DPI queues*), we call them *DPI Head/Tail Pointers*, while the virtual Head/Tail Pointers stored in the reserved DPI memory region refer to the queues in OS memory (*OS queues*), so we call them *OS Head/Tail Pointers*.

Figure 4 illustrates the operation of the descriptor queues on packet reception. The OS preallocates descriptors in the OS queue. A DPI core copies the descriptors to the DPI queue and updates the DPI Tail Pointer, which is visible to the NIC, thus notifying the NIC that descriptors are available for the incoming packets. The NIC copies the received packets, completes the descriptors in the DPI queue, and updates the DPI Head Pointer. At this point the DPI processes the received packets. To allow the OS to consume the received descriptors, the DPI marks the descriptors as complete in the OS queue and updates the OS Head Pointer. Finally, the DPI sends an IPI to an OS core to notify it that reception is complete. The OS can thus proceed to consume the received packets.

In the case of packet transmission, the OS posts descriptors in its transmit queue and advances the Tail Pointer. Ally intercepts the write to the Tail Pointer which is not propagated to the NIC, but to the copy kept in DPI memory, i.e., the OS Tail Pointer. The OS core's MMU raises an IPI to a DPI core, enabling the DPI core to detect that the Tail Pointer has been updated. The DPI copies the newly posted descriptors from the OS queue to the DPI queue. It processes the packets referenced by the descriptors, and updates the DPI Tail Pointer. This Tail Pointer is visible to the NIC, which then fetches the descriptors from the DPI queue and also the corresponding packets. After transmission is complete, the DPI core marks the descriptors complete in the OS queue and raises an IPI to notify the OS core.

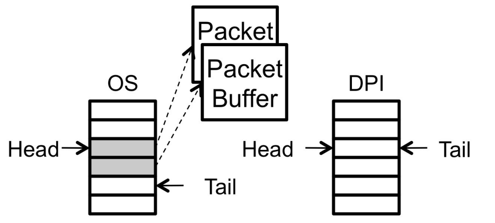
The packet interception mechanism is sufficient to protect low threat workloads on OS cores from inadvertently reading uninspected packet data on the receive path. In order to maliciously transmit uninspected data, the OS core software would need to modify the packet data after inspection completes but before the data DMA to the NIC completes – a time window that cannot be predicted in general.

To provide higher levels of protection, Ally could operate in conjunction with slightly enhanced NIC hardware. Current NICs compute packet data checksums, and could be extended to write checksum values to the receive descriptor queue, and verify DPI-computed checksum values that it reads from the transmit descriptor queue, thereby enabling full detection of packet content tampering. Alternatively, a naive software-only solution to enhance protection for high threat workloads is to copy packet data between OS and DPI memory, incurring higher overheads.

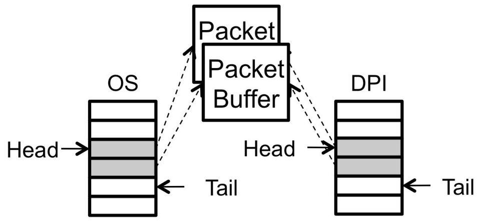
4.4 User-Kernel Space Interaction

Ally supports the execution of a complete operating system in DPI cores with separate kernel-level and user-level execution modes. In this case, the DPI engine can be deployed as a user space application. This has several advantages in terms of programmability, allowing the use of standard libraries and debugging tools, but requires efficient kernel space - user space communication.

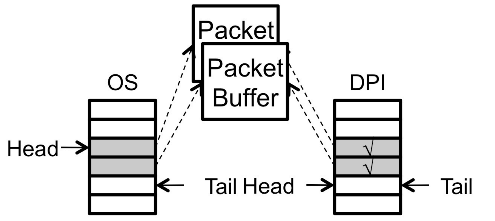
The basic Ally services (packet interception) run in the kernel, while packet inspection is performed in user space. We developed an efficient software mechanism to deliver intercepted packets from kernel space to user space on DPI cores leveraging Ally's operating model. Since all code on DPI cores is trusted, the DPI cores can fetch packet data directly from the kernel-level packet buffers pointed to by the NIC TX and RX descriptors. This is accomplished by mapping the whole kernel space into DPI address space (e.g. using /dev/mem in Linux). In addition, we use a socket-like mechanism (*netlink* in Linux) to communicate updates of the descriptor head and tail pointers to the DPI engine. For example,



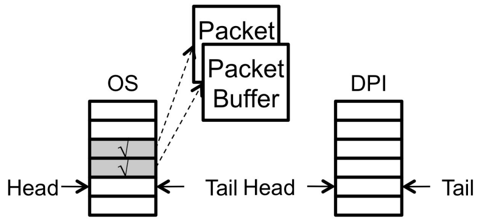
(a) Descriptors are made available by the OS in the OS descriptor queue. OS increases OS Tail Pointer



(b) DPI gets interrupted by the OS core, copies descriptors, and updates DPI Tail Pointer



(c) NIC copies received packets and descriptors, marking descriptors as completed in the DPI queue. NIC updates DPI Head Pointer. **DPI processes packets**



(d) DPI marks descriptors in the OS queue as completed. DPI updates OS Head Pointer. The OS can thus consume the packets and clean the OS queue

Figure 4: Receive: Evolution of the descriptor queues in Ally

when the OS (on the OS core) tries to transmit packets by updating the tail register of the virtual TX queue, the updated value is sent as a netlink message to the user space packet inspection software, which inspects all packets enqueued between the old and new value of the tail register.

5. EVALUATION

We employ a dual strategy to evaluate Ally. First, we validate the functionality of the proposed hardware and firmware modifications by adding Ally enhancements to an x86 full system emulator (QEMU [3]). The modified emulator successfully boots Linux on emulated CPU cores and a separate software environment on a core sequestered for packet processing. Second, to assess the performance of Ally on real hardware, we built a Linux-based prototype. The prototype modifies Linux to emulate the key functionality of Ally needed for packet interception. The resulting system has performance that should approach the performance of real Ally hardware.

For both the QEMU-based and Linux-based prototypes, we use the Netperf micro-benchmarks (`http://www.netperf.org`) as the initial application-level workload running on an OS core. We use the two Netperf streaming tests, TCP_STREAM for transmit and TCP_MAERTS for receive, and the request-reply TCP_RR test. In all cases, a single additional machine running unmodified Linux was used as the network client. Finally, we evaluate Ally performance using SPECWeb2005, a more realistic workload scenario than Netperf in which application processing is significant relative to packet processing.

Table 2: QEMU Results

	TCP_STREAM	TCP_MAERTS	TCP_RR
Instructions/pct	21.33	71.99	246.40
Interrupts/pct	0.04	0.39	1.98
Data movements (bytes/pct)	19.34	31.88	63.05

5.1 Full System Emulation

The QEMU-based system successfully boots unmodified Debian Linux onto a subset of the emulated CPUs, while retaining one core for exclusive use by the DPI engine. As expected, Linux does not detect the existence of the reserved DPI core. The DPI core executes custom code loaded from the BIOS. This custom code intercepts all packets between Linux and the emulated Intel Pro/1000 Ethernet device. After intercepting each packet, the custom code simply forwards it to the NIC or the OS core. To provide insight into the basic cost of packet interception, the custom code on the DPI core was designed only to intercept packets and does not perform actual DPI processing in this initial test.

We extended QEMU to measure significant events for Ally. As shown in Table 2, we measured the baseline overhead of Ally in terms of the number of extra instructions, memory accesses, and interrupts per packet. By packet, we mean an Ethernet frame that is transmitted or received over the external network link. The size of a packet can range from minimum Ethernet frame size to full size 1514-byte frame. All results shown in this Table measure cost on the DPI-core only. Using three Netperf benchmarks, the results indicate that Ally requires at most a few hundred instructions per packet, and tens of bytes of data movement for MMIO and receive descriptors. This overhead would typically be dwarfed by the processing required for an actual DPI application.

5.2 Linux-Based Prototype

To estimate the actual performance of Ally on real hardware instead of QEMU emulated hardware, we extended Linux with low-cost kernel software modifications to emulate key Ally hardware modifications. We chose to emulate Ally using kernel modifications over a architectural simulator for the following reasons. The latency added by the extra hardware is negligible since it only incurs extra latency to MMIO accesses which are trivial compared to the total number of normal memory accesses. Also, architecture simulators are very slow and lack the accuracy of a real system. Therefore, we believe evaluating our queue virtualization approach through software modifications on a real machine is more accurate than on a simulator.

Our prototype extends Linux 2.6.28.7 running on an Intel Core 2 Duo (2GHz) and an Intel PRO/1000 gigabit Ethernet card. The prototype emulates core sequestration by pinning all the OS activity on one core with Linux CPUSETS, and the DPI environment on the other core. It is configured to direct NIC interrupts to the DPI core, and we manually inserted instructions into the NIC driver source code to generate IPIs when virtualized registers are accessed. Another change is to inform Snort from the NIC driver when the DPI core receives IPI. It requires sending Netlink message from kernel to user space. The isolation property is somewhat violated since the DPI core uses the Netlink Service provided by the Linux kernel. However, further investigation shows that it adds almost no overhead to the Linux kernel. In this way, the prototype emulates in software the redirection of the accesses to the NIC registers that would be provided in the Ally hardware by the modified MMU. The prototype provides timing-faithful results not possible to obtain with QEMU.

To evaluate Ally in the context of real packet inspection software, we deployed Snort on the DPI core. Snort is a popular open source Intrusion Detection/Prevention System. All packets intercepted on the DPI core are processed by Snort before being forwarded to the NIC or the OS core. We modified Snort for Ally to use our kernel-user communication mechanism described in Section 4.4.

Although Ally can use an arbitrary number of cores, we limited our experiments to one core for the DPI. This is due to the single threaded nature of Snort. Parallelization techniques could be used to enable multicore scalability of Snort, but we think this is out of the scope of this paper.

We compare our prototype, *Ally*, with two alternative systems. The first one, *Linux*, deploys Snort on unmodified Linux. Linux is configured such that NIC interrupts and Snort are pinned on one core ("DPI core"), while all other activities are pinned on a second core ("OS core"). We stress that this Linux system is not a viable alternative to Ally in practice since it provides no transparency whatsoever. We use it only to verify that Ally does not increase processing costs significantly compared to native Linux, and to evaluate the benefits of Ally's user-kernel interaction mechanism. For the second system, *Xen*, we deployed two virtual machines (VMs), Dom0 and DomU. Each domain has one virtual CPU (VCPU). Snort runs in Dom0, which intercepts all packets to/from DomU. The VCPU running in Dom0 is DPI core while the VCPU running in DomU is OS core. We use core pinning between virtual and physical CPU to avoid VCPU migration leading to performance loss. This setting is arguably the most efficient setting we can use in the current Xen implementation.

Figure 5 shows the throughput achieved by Ally, Linux, and Xen for the Netperf benchmarks. For the streaming workloads, the DPI core is saturated for all three systems and limits the achieved throughput. For Ally and Linux this is mainly a result of choosing a highly CPU-intensive ruleset for Snort as our example use case. Less CPU-intensive packet processing use cases – e.g., passive monitoring, load balancing – would likely achieve full line rate bandwidths. The Ally system achieves 25-32% higher throughput than the Linux system for the streaming benchmarks (TCP_MAERTS and TCP_STREAM), and both systems far exceed the performance of Xen. The low overhead of Ally's packet interception mechanism makes Snort able to consume more cpu cyles when the DPI core is saturated in both cases. Instead, the transaction rate in TCP_RR is limited by the round-trip latency instead of the CPU cycles Snort can consume. The queue virtualization provided by Ally adds a small delay to the transmit and receive paths compared to Linux. As a result, Ally achieves similar but slightly lower TCP_RR transaction rates than Linux, and both systems significantly outperform Xen.

To better understand the reasons behind these high-level performance results, we used OProfile to analyze the breakdown of CPU processing cost on each core, as described next.

5.3 Per-Core Processing Costs

Figure 6 shows the CPU cycles consumed on DPI core and OS core to process each packet on each system for the three Netperf benchmarks. Since Snort is the only user-level process running on DPI core, we further divide the cycles on the DPI core into DPI (user) and

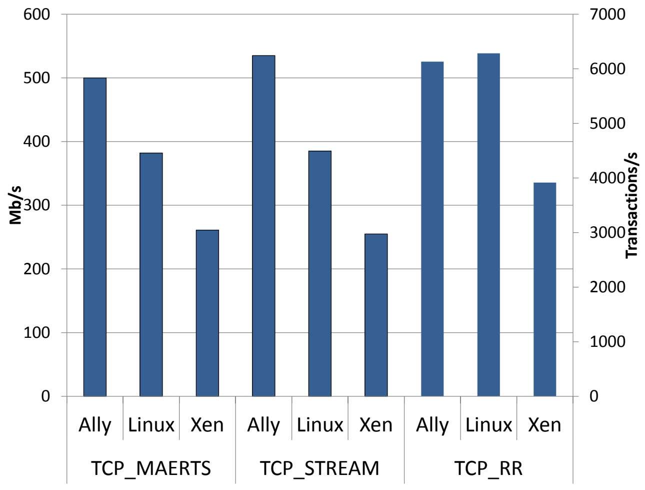


Figure 5: Throughput for Ally, Linux, and Xen running with Snort. Throughput is in Mb/s for TCP_MAERTS and TCP_STREAM, left y-axis, while TCP_RR uses transactions/s, right y-axis

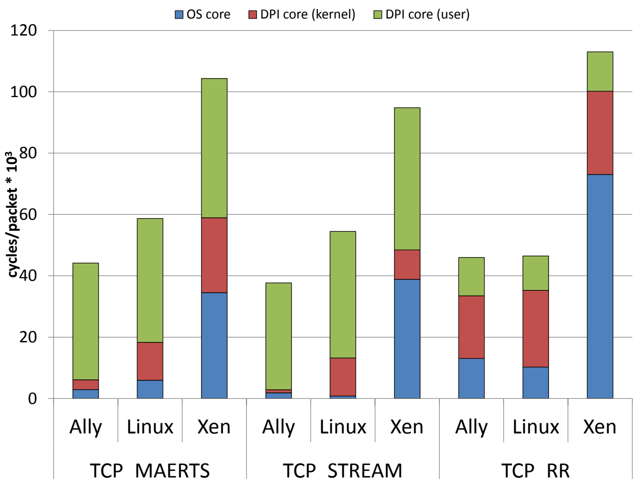


Figure 6: CPU cycles (OS core and DPI core) for Ally, Linux, and Xen running with Snort

Table 3: Classes grouping Linux functions

Class	Description
Packet Interception	Functions used to intercept packets
Memory copy	Functions used to copy data between kernel and user space
Network	Transmit/receive path related functions
Hypervisor	Functions in Xen
Other	e.g. time, scheduling

DPI (kernel) to explicitly show the Snort overhead. The results show that Snort consumes a similar number of cycles per packet across the three systems. This implies that Snort is inspecting about the same amount of data for each packet in all three systems. Further studies (not shown) revealed that most cycles are consumed by Snort's pattern matching algorithm, which is used to compare the packet payload against all rules. Snort consumes the majority of CPU cycles in TCP_MAERTS and TCP_STREAM since many packets with large payload are inspected. In contrast, for TCP_RR, only one small packet is inspected at a time.

The results also show that the three systems have significantly different processing costs for DPI (kernel) and OS core. In particular, Ally and Linux have roughly similar OS core costs which are much lower than for Xen. Ally has lower DPI (kernel) costs than Linux and Xen for all three benchmarks.

To analyze these differences, we need a further breakdown of processing costs for DPI (kernel) and OS core. We thus group source code functions into the cost categories shown in Table 3. Functions are placed into these categories based on a static analysis of the Linux source tree and on an analysis of the dynamic call graph of kernel execution. The following subsections break down the cost of DPI (kernel) and OS core according to these function categories.

5.3.1 DPI Core Costs

Figure 7 shows the breakdown of DPI (Kernel) cost per packet using the categories of Table 3. We are interested in the comparison of packet interception mechanism between Ally and Linux and their corresponding costs on memory copying. In Linux, Snort normally uses the libipq mechanism to fetch packets from kernel space. This mechanism leverages the Linux iptables mechanism to intercept and enqueue network packets at kernel-level into a special queue. From there, the packets are delivered to the user-level Snort application using the Linux netlink mechanism, which provides a standard socket interface for user-kernel communication. Each netlink message contains one packet and is copied from kernel space to user space. After inspection, Snort sends a verdict back into the kernel specifying how to deal with those packets (ACCEPT or DROP). Packets accepted by Snort continue to traverse the network stack. This libipq mechanism is used in the Linux and Xen systems.

As shown in Figure 7, Linux has higher packet interception overhead than Ally for the streaming workloads. There are two reasons. First, libipq has a more complex API than our queue virtualization approach. Second, for libipq, the number of netlink messages is equal to the number of packets that are analyzed by Snort. Therefore, Linux has similar netlink overhead per packet in each benchmark. For Ally, the number of netlink messages corresponds to the updates of tail/head pointers, which for the streaming benchmarks is much less than the total number of packets, leading to lower overhead for these workloads than in Linux. In addition, Ally has the lowest overhead in memory copy

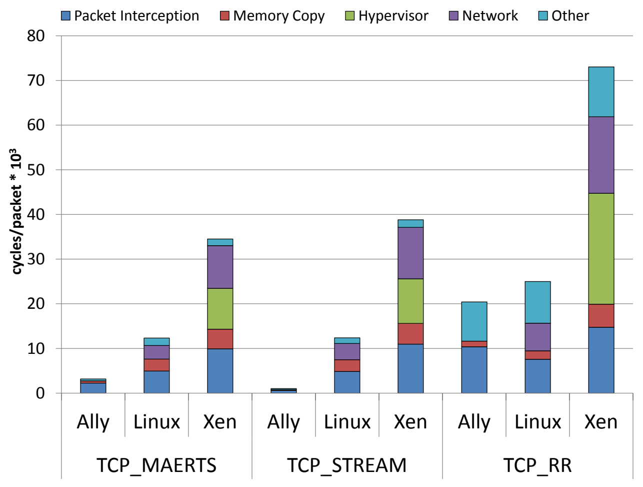


Figure 7: CPU cycles for DPI core (Kernel) for Ally, Linux, and Xen

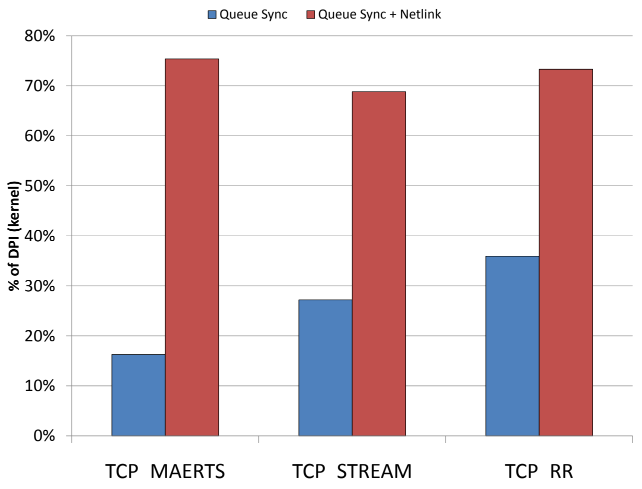


Figure 8: Overhead of user space communication in Ally

category. The reason is that in Linux the libipq uses the netlink mechanism to transport entire packets to user space whereas in Ally the netlink connection is only used to transport the descriptor head and tail pointers to Snort and all packet data are directly accessed via memory mapping. Ally does not have any overhead in network category in all benchmarks. This indicates the overhead incurred on DPI core is purely for packet interception. We are unable to completely separate the packet interception mechanism and the network transmission/receiving path into two different cores in Linux and Xen. [FIXME: need a better way to describe this]

For TCP_RR benchmark, the packet interception overhead of Ally exceeds that of Linux. There are two reasons. First, the updates of tail/head pointers are equal to the number of packets since only one packet is sent or received at a time. Therefore, the number of netlink messages in Ally is similar to that in Linux. Second, the queue virtualization requires updating the real descriptor head and tail which is time consuming MMIO access and cannot be amortized by many packets as in the streaming benchmarks. We do not count MMIO accesses in Linux or Xen since they are not part of libipq and should belong to the network transmission/receiving path.

In all cases Xen has much higher DPI core processing cost than Ally and Linux. Most of the overhead in hypervisor is due to the expensive grant table mechansim which is used to remap packets from one domain to the other. Even the network processing within Dom0 greatly exceeds that of Linux and Ally because of the complex backend driver and bridge mechanism added in Dom0. Overall this result reflects a relatively high cost of packet interception and network interface virtualization in the current Xen implementation. However, virtualization costs might be reduced by future enhancements to Xen, and optimizations similar to those we added to Ally could potentially be applied to Dom0 to reduce the cost of intercepting and sending packets to the user-level Snort application in Dom0, but this is out of scope of this paper.

The DPI (kernel) cost in Ally has three components: 1) NIC queue virtualization and synchronization, 2) Our netlink-based mechanism for communicating with Snort, and 3) Snort invoked system calls and other miscellaneous kernel costs. We ran experiments with Ally in different configurations to measure the contributions of each component on overall DPI (kernel) cost. In addition to the full configuration that was used in the previous experiments, we ran experiments with the DPI core performing only queue virtualization without Snort, and we ran experiments with the DPI core performing queue virtualization together with our netlink-based mechanism to communicate with the Snort user process but without any rule processing in Snort. The results in Figure 8 show that queue virtualization accounts for 15-36% of DPI (kernel) processing costs, and the combination of queue virtualization and communication with the user level process accounts for 68-75% of DPI (kernel) cost. The remaining 25-30% of total DPI (kernel) cost is due to application-invoked (Snort) system calls (e.g., to timestamp packets) and miscellaneous kernel costs.

[FIXME: The following paragraph looks redundant. Need to extract some useful information from there]

5.3.2 *OS Core Costs*

The results show that OS core costs with Ally are similar to the costs with unmodified Linux. In fact the costs with Ally are slightly lower. However the difference is less than 15%. We found that there are

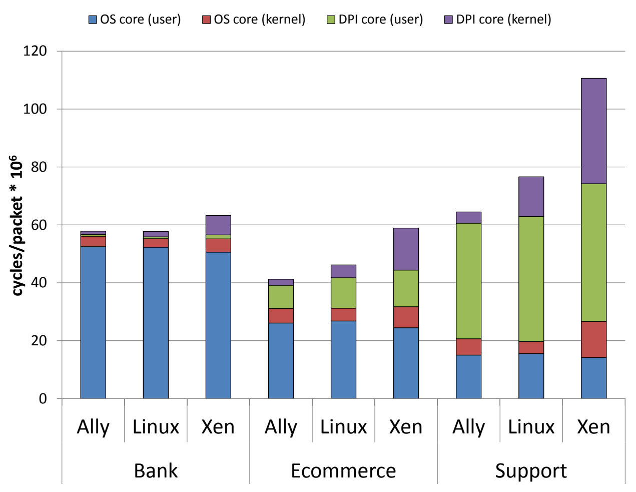


Figure 9: Processing costs for SPECWeb2005 workload

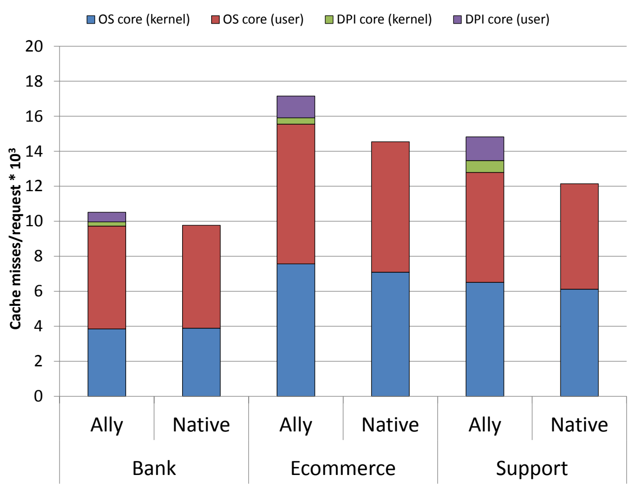


Figure 10: Cache misses for the SPECWeb2005 workloads

several reasons contributing to the difference. First, in Ally, the OS core accesses virtualized NIC registers rather than performing the MMIO accesses which incur higher latency. (Ally performs these MMIO accesses on the DPI core, and their cost is reflected in the results of Figures 7 and 8). Second, Ally incurs a little more overhead in netlink functions due to the service shared by the DPI core. Lastly, for the receive-heavy TCP_MAERTS workload in Ally, the DPI core processing can fetch packet data into the processor cache hierarchy in advance of OS core processing of the same packets, potentially reducing the cost of OS core processing.

5.4 SPECWeb Results

While Netperf is useful for understanding system behavior in network-intensive scenarios that place high requirements on packet processing services, most real workloads present a more balanced mix of computing and networking. We ran experiments using SPECWeb2005 to evaluate Ally with a realistic workload presenting a mix of requirements for each resource type. Each benchmark – Bank, Ecommerce and Support – is run with 100 simultaneous sessions. Apache is run on OS core (user) while Snort is run on DPI core (user).

The results in Figure 9 shows that Support has the highest number of packets per request which reflects in the cost of Snort. The traffic encryption in Bank makes Apache consume much more cycles among all cases. The kernel cost in Ally shows that its packet interception mechanism is more scalable than Xen in which case the hypervisor cost increases a lot under higher number of packets in Support.

To study the potential cache sharing and interference effects, we run Apache with and without Snort using the SPECweb workloads. Results in Figure 10 show that Apache has a similar number of cache misses running with or without Snort. The cache misses of OS core (kernel) in Ally are a little more than in Linux possibly because of resource contention due to the Netlink service shared by the DPI core. The majority of other misses are due to kernel page allocation. The cache misses of Snort ("DPI core (user)") are mainly due to packet decoding and payload inspection, since received packet data is first touched in our solution by Snort rather than the Linux kernel (zero-copy for inspection). The DPI core (kernel) cache misses in Ally are due to descriptor queue synchronization, as descriptors are invalidated in the cache when the NIC DMAs the modified descriptors into the main memory. We also found that there is almost no difference between the average request latencies of the two cases. The main contribution to the latency is Apache, not of the packet processing in Snort.

6. RELATED WORK

Conventional DPI appliances, and other network packet processing appliances such as firewalls, are typically implemented as special-purpose devices [7]. Ally enables moving this functionality to standard server platforms, potentially removing central network bottlenecks and replacing them with distributed local processing in software. Previous work advocated implementing these functionalities in virtual machines (VMs) running on a hypervisor along with application VMs, and using Trusted Computing (TC) hardware in modern processors to ensure that the I/O appliances are running authorized code [2]. Ally differs from these approaches by eschewing the need for a hypervisor at all. By achieving complete independence from host software, Ally provides a general solution that works seamlessly for customers with diverse OSes and hypervisor deployments in a shared datacenter. Moreover, the core sequestration model avoids the complexities of constructing chains of attestation using TCB.

In the context of business laptops, the Intel Vpro technology [8] provides limited packet filtering capabilities in the NIC. Ally provides a much more flexible and powerful infrastructure to allows effective packet analysis.

Similarly to Ally, a hypervisor could be used to provide system partitioning (OS-partition and DPI-partition). Hypervisors can additionally host virtual machines called driver domains to transparently intercept data exchanged between running guest VMs and I/O devices [9, 10]. Additional packet processing such as DPI could be performed in software in these driver domains [11]. In comparison to using software virtualization for partitioning and I/O interception, the hardware/software approach provided by Ally has the conceptual advantage that the "OS-partition" can run arbitrary user code. In particular, the OS-partition can itself run a hypervisor. Without hardware partitioning, this would require full support for nested layers of virtualization. Recent work indicates that such nesting may be feasible with reasonable performance degradation [12]. However, this solution achieved good performance for I/O only by bypassing the virtualization layers, precluding packet interception. In addition, the hardware-based isolation mechanisms used in Ally are arguably more resistant to attacks compared to potentially buggy hypervisor software. This is the case even if Trusted Computing [13] techniques are used to verify the authenticity of binary executables. Some researchers have argued recently that hardware partitioning offers a variety of significant advantages over software hypervisors [14].

Some techniques that Ally uses for lightweight packet interception could also be adapted for driver domains in software virtualized environments. For example, a running guest VM that has direct access to a NIC (or virtual context of a NIC) could be transparently switched on-the-fly by the hypervisor to use a driver domain that intercepts and inspects packets by virtualizing the NIC queue using similar techniques as Ally.

The BitVisor hypervisor aims to provide transparent I/O services for a single guest VM using device queue virtualizations that are similar to our approach [15]. BitVisor differs from Ally in not using hardware core sequestering, and therefore cannot support a full operating system and application stack on a dedicated DPI core, or run a hypervisor with multiple guest OSes on the OS cores.

The use of a high privileged core for security and monitoring has been proposed for a system called INDRA [16] and by Chen *et al.* in the context of log-based architectures [17]. In these systems, the privileged core is called "resurrector" or "lifeguard". Ally extends this functionality providing efficient I/O analysis capabilities to the privileged core.

7. CONCLUSIONS

Ally enables software-independent and transparent deployment of packet processing services like DPI on a multicore processor. Our evaluation on a system emulator and a prototype Ally system demonstrate the feasibility of the Ally approach and analyze in detail the contributions of the design components to overall performance. Our results demonstrate that Ally is competitive in efficiency with deploying DPI non-transparently on native Linux, and is far more efficient than a system that uses Xen software virtualization to achieve transparent packet inspection.

We plan to extend this work to support and evaluate packet processing services using emerging multi-context NICs (PCIe SR-IOV NICs) [18]. Given the low cost of queue virtualization observed in our current prototype, it is highly likely that the cost of Ally's interception mechanisms will be a function dominated by the the link bandwidth or packet rate, and not by number of contexts. In addition, we plan to evaluate Ally for larger-scale multicore systems. Ally should easily leverage future processors equipped with several tens of cores by distributing the packet inspection engine.

Ally and complementary technologies are bringing greater packet processing capabilities to general purpose servers with powerful programmable CPUs and large amounts of memory, enabling new rich network services. This motivates many potential future research investigations in the context of large datacenters to study problems including secure deployment, configuration, and coordination of an ensemble of Ally instances and services.

8. REFERENCES

[1] Deep packet inspection: 2009 market forecast. *Lightreading Insider*, 8(11), December 2008.

[2] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. Ettm: a scalable fault tolerant network manager. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 2011.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41. USENIX Association, 2005.

[4] Intel Active Management Technology. Online: http://www.intel.com/technology/platform-technology/intel-amt.

[5] HP Integrated Lights-Out (iLO) Standard. Online: http://www.hp.com/go/ilo.

[6] Aravind Menon, Simon Schubert, and Willy Zwaenepoel. Twindrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest os drivers. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 301–312, New York, NY, USA, 2009. ACM.

[7] CloudShield Technologies. Online: http://www.cloudshield.com.

[8] Intel Vpro Technology. Online: http://www.intel.com/vpro.

[9] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[10] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, Andrew Warfield, and Mark Williams. Safe hardware access with the Xen virtual machine monitor. In *OASIS '04: Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT Infrastructure*, October 2004.

[11] D. McAuley and R. Neugebauer. A case for virtual channel processors. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 237–242, New York, NY, 2003. ACM.

[12] M. Ben-Yehuda, M. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. Yassour. The turtles project: Design and implementation of nested virtualization. In *OSDI 2010*.

[13] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, 2008.

[14] E. Keller, J. Szefer, J. Rexford, and R. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[15] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. Bitvisor: a thin hypervisor for enforcing i/o device security. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, 2009.

[16] Weidong Shi, Hsien-Hsin S. Lee, Laura Falk, and Mrinmoy Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 102–113, 2006.

[17] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. Gibbons, T. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, 2008.

[18] Y. Dong, Z. Yu, , and G. Rose. SR-IOV networking in Xen: Architecture, design and implementation. In *WIOV '08: Proceedings of the 1st Workshop on I/O Virtualization.*