Improving Virtualization in the Presence of Software Managed Translation Lookaside Buffers

Xiaotao Chang[1], Hubertus Franke[2], Yi Ge[1], Tao Liu[1], Kun Wang[1], Jimi Xenidis[3], *
Fei Chen[1], Yu Zhang[1]

[1]IBM Research-China
Beijing, China
{changxt, geyi, liuttao, wangkun, uchen, zhyu}@cn.ibm.com

[2]IBM Thomas J. Watson Research Center
New York, USA
frankeh@us.ibm.com

[3]Qualcomm Research Silicon Valley
California, USA
jimixl@qti.qualcomm.com

ABSTRACT

Virtualization has become an important technology that is used across many platforms, particularly servers, to increase utilization, multi-tenancy and security. Virtualization introduces additional overhead that often relates to memory management, interrupt handling and hypervisor mode switching. Among those, memory management and translation lookaside buffer (TLB) management have been shown to have a significant impact on the performance of systems. Two principal mechanisms for TLB management exist in today's systems, namely software and hardware managed TLBs. In this paper, we analyze and quantify the overhead of a pure software virtualization that is implemented over a software managed TLB. We then describe our design of hardware extensions to support virtualization in systems with software managed TLBs to remove the most dominant overheads. These extensions were implemented in the Power embedded A2 core, which is used in the PowerEN and in the Blue Gene/Q processors. They were used to implement a KVM port. We evaluate each of these hardware extensions to determine their overall contributions to performance and efficiency. Collectively these extensions demonstrate an average improvement of 232% over a pure software implementation.

1. INTRODUCTION

Virtualization has seen increased popularity due to cloud computing, deployment of virtual appliances and consolidation. In contrast to standard system deployment where a single operating system (OS) controls the physical machine it is running on, virtualization enables multiple OSs to run concurrently on a single physical machine. Doing so, enables new important features. First, different types and versions of OSs can run side by side. For instance, legacy applications relying on older versions of an OS can be deployed at the same time as modern applications that require more recent OSs. Furthermore, applications and their OSs can be isolated, which gives rise to the notion of software appliances. Second, probably equally important, is that this collocation enables a much higher utilization of computing resources. It is known that infrastructure services such as print and file servers, as well as end-user desktop systems, demonstrate poor system utilization that often does not reach a double digit percentage [3]. Virtualization, with the resulting ability to collocate OSs, enables a significantly higher utilization and is one of the principal enabling technologies that has been driving cloud computing.

Virtualization introduces a new layer of system software, called the hypervisor (or Virtual Machine Monitor, VMM), that controls the access to the physical resources from guest OSs. The guest OS has restricted control of the physical resources and has to engage with the hypervisor to get access. This new layer results in additional overhead in the execution of code and accesses to the I/O devices that can grossly be categorized in memory management, interrupt handling and I/O accesses. Processor technology has advanced and now provides many techniques to mitigate some of this overhead.

In this paper we pay close attention to the impact on the memory management subsystem under virtualization. For instance, almost all modern processors are equipped with TLBs. Two principal mechanisms for handling TLB misses are commonly found in modern architectures, namely hardware managed TLBs or software managed TLBs, or some combination of the two. Processors with hardware managed TLBs automatically walk the page table to locate a valid page table entry for the specified virtual address. If an entry exists, it is brought into the TLB and the translation is retried: this time the access will hit in the TLB, allowing the program to proceed. For example, Intel and AMD's x86 processors, IBM's Power Server ISA and ARM's Cortex are all equipped with hardware managed TLBs. With the introduction of virtualization, address translation is a dual lookup, one within the guest translation and one within the hypervisor/host environment. In systems with hardware managed TLB, the TLB implementation can be transparently extended without impact on the system stack.

In contrast, in processors with software managed TLB, a TLB miss generates a "TLB miss" exception, and OS

*All authors were with IBM when working on this paper.

ISCA '13 Tel-Aviv, Israel

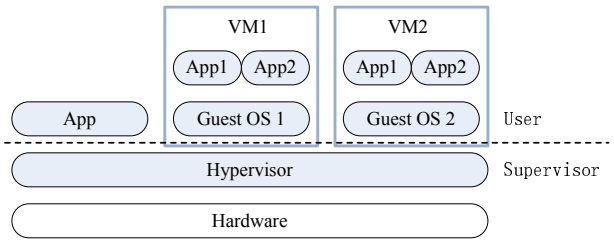


Figure 1: Pure Software Virtualization on Traditional Power Embedded Core

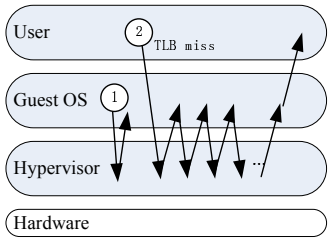


Figure 2: Interrupt Handling on Pure Software Virtualization

code is responsible to traverse the page table to identify the translation in software. The OS then loads the translation into the TLB and restarts the program from the instruction that caused the TLB miss. Processors such as MIPS, UltraSPARC, and Power Embedded ISA are equipped with software managed TLBs. With virtualization, two translations need to be done in software, which are performed in different protection domains.

In this paper, we describe our design of hardware extensions to support virtualization in systems with software managed TLBs. These extensions were implemented in the Power embedded A2 core, which is used in the IBM PowerEN[10, 11] and the IBM Blue Gene/Q[14] processors, and subsequently utilized for a port of a Kernel-based Virtual Machine (KVM) hypervisor. A performance comparison between a KVM implementation and a pure software mechanism is conducted which shows 232% performance improvement on average.

The rest of this paper is organized as follows. In Section 2, we describe the pure software based classical virtualization techniques on the Power embedded core which also highlights some of the inefficiencies of the approach. Section 3 proposes the hardware assisted virtualization techniques for the Power embedded core, which is a software managed TLB processor. Section 4 describes the virtualization implementations with different hardware extension mechanisms on Power embedded core. Section 5 presents the experimental results and evaluates the performance impact of the different hardware extensions. Section 6 summarizes related work and Section 7 concludes.

2. CLASSICAL VIRTUALIZATION MECHANISM ON TRADITIONAL POWER EMBEDDED CORE

Popek and Goldberg's 1974 paper [22] defines requirements for what is termed classical virtualization. By their standards, a piece of software can be considered a hypervisor if it meets the following three requirements[1]: fidelity, performance and safety.

The trap-and-emulate technique for long was thought to be the only practical method for virtualization. We refer to it as EMUL virtualization throughout this paper, as compared to the hardware assisted virtualization.

As shown in Figure 1, in the traditional Power embedded core architecture, two operating states or modes are defined: supervisor (privileged) and user (non-privileged). An attempt to execute a privileged instruction or to access a privileged register while in user mode causes a Privileged Instruction exception to occur.

In EMUL virtualization, all instructions of the guest machine (aka virtual machine, VM), including ones executing in the guest OS, are executed in user mode. All privileged instructions running in the guest machine will trap to the hypervisor (supervisor mode) and then the hypervisor performs the emulation and returns to the guest. Usually, because there are a significant number of supervisor instructions and privileged register accesses that need to be trapped and emulated, EMUL virtualization introduces significant overhead on the Power embedded core. For example, the privilege instruction *'mtmsr r0'* (move the MSR to GPR r0), which is utilized for monitoring processor state and interrupt handling, issues 669 host instructions during EMUL virtualization. This overhead includes the guest state saving (158 instructions), the hypervisor management (144 instructions), the emulation (290 instructions) and the guest state restore (77 instructions). Different interrupts can cause various overheads. In this paper, the detailed instruction counts were collected using the Mambo [9] full-system simulator, and all other performance numbers were collected on real hardware.

Next, we will examine the specifics of the software virtualization approach for the traditional Power embedded core.

2.1 Interrupt Interception

In the Power embedded core, the location of the interrupt service routines is defined by a special purpose register (SPR), named IVPR (Interrupt Vector Prefix Register). Since the guest OS is always executing in user mode, the hypervisor needs to redirect the IVPR register and point it to the address of the hypervisor interrupt routines before guest machines run. Usually, there are three operations that need to be performed in the interrupt routines: hypervisor mode switch, privileged instruction or function emulation, and interrupt injection to the guest. Hypervisor mode switch implies a switch in privilege level between guest and hypervisor, saving the context of guest machine and preparing the virtualization related guest data structures. All of the guest related states including general registers and some special registers, such as PID (Process ID), PC, stack pointer, return address, etc., are saved to a reserved data structure (called vcpu). The IVPR is also redirected to the interrupt routines of the host OS.

There are two types of instructions that need to be emulated. The first is the set of privileged instructions that can not execute in user mode. The hypervisor emulates these instructions and returns the results to the guest. The second is the set of instructions which access privileged hardware resources, such as some SPRs, TLB etc. The hypervisor maintains a shadow copy of such privileged resources in vcpu. All

accesses to the privileged resources in the guest machine are directed to the shadow copy.

According to the type of interrupts, the hypervisor takes corresponding actions. As we described above, privileged instructions are emulated in a hypervisor environment, shown in Figure 2 as type-1. If an interrupt is destined to the guest OS for further processing, the hypervisor will inject the interrupt context into the guest OS and switch to the guest OS's interrupt routines, shown in Figure 2 as type-2.

2.2 MMU Emulation

In a virtualized system, besides the guest OS maintaining page tables for guest virtual page number (GVN) to guest physical page number(GPN) mappings, the hypervisor also maintains mappings of GPN to host physical page number (HPN).

In a software managed TLB processor, privileged system code (either OS or in the case of virtualization, the hypervisor) uses privileged instructions to maintain the TLB. Hence, in the EMUL virtualization approach, all TLB instructions, such as TLB read/write/search operations, will invoke a trap and be emulated by the hypervisor. To accomplish the emulations, the hypervisor maintains a shadow TLB structure, which is attached to the guest machine for its TLB related operations. The shadow TLB contains all valid TLB entries seen by the virtualized guest OS. When an address translation for instruction fetch or data access misses in the hardware TLB, a TLB miss exception occurs which is intercepted by the hypervisor. The hypervisor first conducts a lookup in the shadow TLB. If a corresponding valid shadow TLB entry is found, a mapping from GVN to GPN can be extracted from the entry. Then the hypervisor translates the GPN to an HPN and loads the new address mapping ($GVN \rightarrow HPN$) into the hardware TLB entry. If no entry is found in the shadow TLB, which is a virtual TLB miss of the guest machine, the hypervisor will inject the exception into the guest machine and relinquish control to the guest OS handler to process this exception with guest page tables. The guest OS handles the TLB miss by issuing TLB read/write/search operations, all of which have to be intercepted and emulated. As a result, the resolved address mapping ($GVN \rightarrow GPN$) information is saved into the shadow TLB for subsequent searches. Finally, the TLB miss is handled and the guest machine can proceed. This process is shown as type-2 exception in Figure 2.

To guarantee the security and resource management requirements, two mechanisms are required in emulating the TLB access and lookup operations in Power embedded core: address space (AS) isolation and PID mapping. The Power embedded core provides two address spaces for instruction and data access: address space 0 and address space 1. Generally the host system runs in address space 0. Thus address space 1 can be reserved for guest machines. This is done by setting a reserved bit in MSR and a reserved bit in the TLB entry. The former is used to identify the target address space of memory accesses. The latter is applied to indicate the address space where address translation occurs. The precondition of this method is to guarantee that the reserved bit is not used in the guest machine. This is true for many popular OSs such as Linux and BSD. When a TLB lookup operation is performed, the reserved bit in the MSR is compared with the corresponding bit in the TLB entry, which guarantees the address space isolation between guest

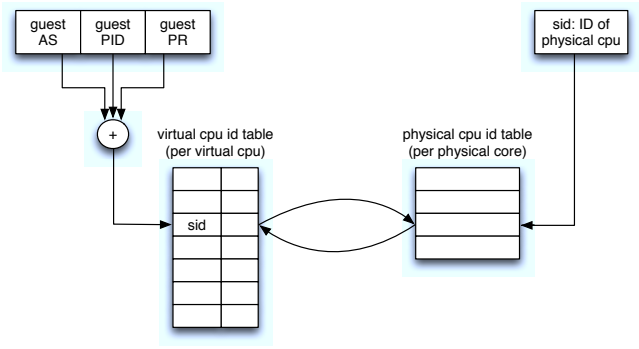


Figure 3: PID Mapping Mechanism for Virtualization

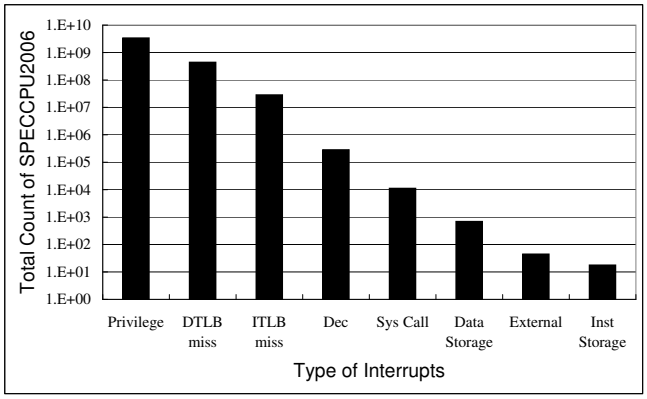


Figure 4: Total Traps to Hypervisor of SPECCPU2006 Test Set

machine and host machine[8].

In EMUL virtualization, all isolations or partitions between different guest machines are implemented by the PID mapping. PID is a unique value for each process allocated by the OS, and is a part of a virtual address. Each TLB entry has a PID field to indicate which virtual address space it belongs to. All guest processes, whether the guest OS (supervisor privileged) or guest applications (user mode), are user-level processes of the host machine. Though the guest OS allocates a unique PID for each process, this PID may not be globally unique across all guest machines. Consequently the hypervisor must keep a globally unique PID for each guest process. That is accomplished by an additional PID mapping table per guest (virtual cpu id table), which maps guest's virtual PID to real PID (sid) that is ultimately loaded into the hardware register, as shown in Figure 3. This additional lookup during hypervisor mode switch further increases the overhead of EMUL virtualization.

2.3 Performance Overhead in EMUL Virtualization

To determine the overhead of pure software based EMUL virtualization mechanism, we conducted a performance study running SPECCPU2006 [23, 19] within an OS that was not virtualized, and then in a virtualized OS. The platform was an embedded PowerPC, PowerEN processor, which will be described in detail in Section 4.

Benchmark	Privilege	DTLB Miss	ITLB Miss	Performance Degradation
perlbench	237559	28475	20443	68%
bzip2	301906	49259	2797	87%
gcc	298313	38782	13669	86%
mcf	363920	51847	2848	99%
milc	390424	44498	2818	97%
gobmk	317475	36108	3458	82%
hmmer	6889	793	444	1%
sjeng	346689	39779	4365	93%
libquantum	10818	1135	555	2%
h264ref	270918	36637	4941	78%
lbm	355284	40400	2106	92%

Table 1: Trap Rate per Second Breakdown with SPECCPU2006 in EMUL Virtualization

Firstly, we counted and listed all different guest exceptions that trapped to the hypervisor in SPECCPU2006 cases, as shown in logarithmic coordinates in Figure 4. We found that three kinds of exceptions dominate (more than 99.9%) in guest exceptions: privileged instruction exception, DTLB miss exception and ITLB miss exception. Hence the discussion below will focus on these three exceptions. Table 1 shows the relative performance degradation and the rate per second of exceptions on each benchmark. The performance degradation is on average 72% with a maximum of 99%. Note that the degradation is not linearly correlated to the number of traps since various emulations require significantly different number of instructions. For instance DTLB misses require between 1K and several K-instructions to emulate. There are two different kinds of DTLB traps, *minor faults* and *major faults*. A minor fault occurs when a $GVN \rightarrow GPN$ mapping exists in the shadow TLB, which can conduct a write to the hardware TLB after the GPN to the HPN translation by the hypervisor. A major fault occurs when no mapping is found in the shadow TLB and the miss needs to be relegated to the guest OS for resolution, which will potentially cause a series of privileged traps.

Focusing on these three exceptions guided our work to investigate, propose, implement and evaluate methods that allow for a more efficient virtualization thus reducing the need for constant trap-and-emulate.

3. HARDWARE ASSISTED VIRTUALIZATION DESIGN ON POWER EMBEDDED CORE

In this section, we describe the design of hardware extensions to support virtualization in systems with software managed TLBs to overcome the inefficiencies identified in the previous section.

3.1 Guest Mode Support

As illustrated in the previous section, the privileged instruction trap causes performance overhead because the guest machine always executes in user mode in EMUL virtualization. If the hardware was able to execute guest privileged instructions directly, many privileged instructions would not cause exceptions anymore.

To accomplish this, an additional instruction privilege level is introduced, leading to 3 levels: guest user (applications), supervisor (guestOS), and hypervisor privileged, shown in Figure 5, just like previous processors such as ARM Cortex A-15[4] and Freescale e500mc[12] did.

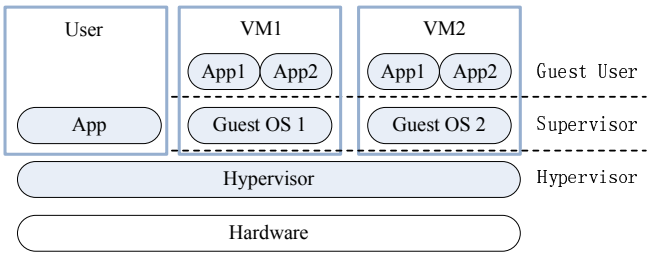


Figure 5: Virtualization Levels on Hardware-Assisted Power Embedded Core

All privileged instructions are divided into hypervisor privileged instructions and supervisor privileged instructions. With the support of guest state, all supervisor privileged instructions can be executed directly on the hardware if the guest machine is running in supervisor state, which implies there is no need to trap-and-emulate. The hypervisor privileged instructions running or accessing a hypervisor privileged resource (such as some SPRs) in guest state or supervisor state still lead to a hypervisor privileged exception.

3.2 Logical Partition Identification

Guest machine isolation is one of the key characteristics that the hypervisor and the virtualization technology must guarantee. As mentioned in the previous section, in EMUL virtualization, the hypervisor uses the PID mapping mechanism to isolated different virtual machines. The PID mapping is implemented by trap-and-emulate on PID access instructions, which means the PID has to be a hypervisor privileged resource and access to PID still leads to a privileged exception.

The guest state support is insufficient to build a globally unique PID space. To eliminate the mapping overhead, the hypervisor privileged Logical Partition Identification Register (LPIDR) is introduced to identify a guest machine. The combination of LPID and PID is a unique value in the whole system for each process and is applied to the address translation in the processor with guest state support. The $< LPID, PID, vaddr >$ is maintained in the TLB. With the presence of the LPIDR, the mandatory PID mapping, that had to be maintained in the EMUL implementation, is not necessary. The TLB management instructions can be executed in supervisor state and the hardware transparently extends the $< PID, vaddr >$ with the LPID when writing it to the TLB.

We analyzed the programs of the SPECCPU2006 C test set with respect to the number of exceptions during their executions. On average, the number of hypervisor privileged exceptions is only 10.2% of the number of privileged instruction exceptions in EMUL virtualization. As shown in the Figure 6, about 90% overhead caused by the privileged instructions could be reduced with the help of Guest Mode and LPIDR.

3.3 Interrupt Redirect

As exceptions and interrupts are treated by the architecture and the system software in the same manner, we do not distinguish them and use these terms interchangeably. In-

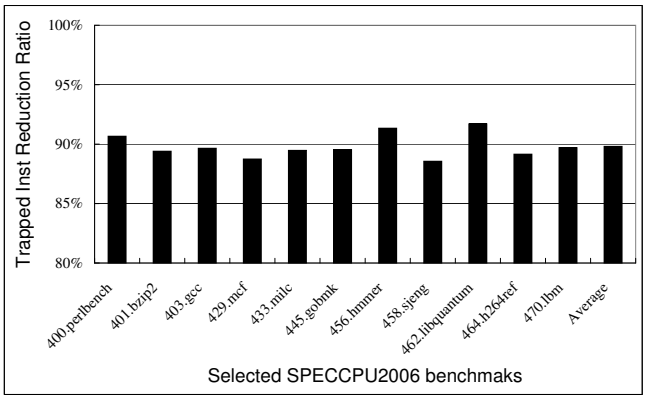


Figure 6: Trapped Instructions Reduction Ratio with Guest Mode and LPIDR

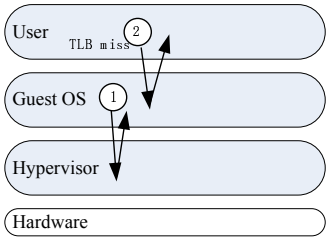


Figure 7: Interrupts Handling on Hardware-Assisted Virtualization

terrupts in Power embedded cores use the IVPR as the base address of the interrupt handler vector. Interrupts that are directed to the hypervisor state use registers SRR0 (for the instruction address) and SRR1 (for the MSR) to save the context of interrupts, ESR (Exception Syndrome Register) to post exception syndrome information, and DEAR (Data Exception Address Register) to keep the virtual address of a data reference.

The overhead for a hypervisor mode switch between the guest and the hypervisor on Power embedded cores is 235 native instructions, including 158 for guest data saving and 77 for hypervisor context restore respectively. Intuitively, if interrupts can be directed to the guest handler upon exception without a trap to the hypervisor, the overhead for hypervisor mode switches between the guest and hypervisor can be eliminated, as the type-2 shown in Figure 7. Not every type of exception can be redirected though, for example, the hypervisor privilege exception and the I/O interrupt. However, most of the TLB miss exceptions which are also responsible for most of the performance overhead can be redirected because they need the intervention of the guest OS.

Hence, a new register is introduced in the Power embedded core, named GIVPR (Guest Interrupt Vector Prefix Register), which is used to hold the base address of guest interrupt handlers. All TLB related exceptions (I/D TLB miss exception and I/D TLB storage exception) can be configured by the hypervisor to redirect to the interrupt handlers of the guest machine.

Although the interrupt redirect mechanism can skip the overhead for hypervisor mode switch, TLB related exception handlers still need many hypervisor interventions. For example, some SPRs (Special Purpose Register), such as SRR0/1, ESR, DEAR etc., are accessed in interrupt handlers and always hypervisor privileged, which means access to them needs trap-and-emulate and the hypervisor maintains their shadow copies.

3.4 Privileged Resource Mapping

To further improve performance of guest machine, a register mapping mechanism is introduced in Power embedded cores to build a guest copy of frequently used SPRs in hardware, and redirect the SPR access in guest machine to them. This redirection takes place when executing in supervisor state. For example, when an instruction *'mtspr SRR0,r5'* is executed in supervisor state, the access to SRR0 is redirected to GSRR0, instead of accessing SRR0. The mapping is transparent to the guest machine, and separates performance critical resource of the hypervisor and the guest OS. It removes the requirement of the hypervisor software to handle the interrupts with these SPR accesses, and the needs to maintain shadow copies for guest machine. In addition, it keeps the same programming interface for both host and guest OS. Once register mapping is implemented, the accesses to the corresponding registers are supervisor privileged since the guest kernel can access them natively.

3.5 Instruction Mapping

The register mapping above is utilized for instructions that access registers directly, such as *mtspr* and *mfspr*. In Power embedded cores, the *rfi* instruction (return from interrupt) is used to return from a base class interrupt. All interrupt handlers use it to return to the interrupted program. Its semantic is as follows: $MSR \leftarrow SRR1$ and $PC \leftarrow SRR0$. When executing in guest machine, execution of an *rfi* instruction can be mapped to *rfgi*, which is executed in place of the *rfi*. The semantic of *rfgi* is below: $MSR \leftarrow GSRR1$ and $PC \leftarrow GSRR0$.

This mechanism is called instruction mapping, which allows the elimination of trap-and-emulate overhead for *rfi* instruction and the same programming interface for guest and hypervisor.

3.6 MMU Virtualization

With the above mechanisms, the number of trap-and-emulates for TLB misses can be significantly reduced, but the traps for TLB management instructions such as *tlbwe* (write) and *tlbre* (read) still exist.

When a *tlbwe* instruction is executed, an address mapping entry is filled into the TLB. Because the guest machine can not directly operate on the hardware TLB, and the guest has no knowledge of the mapping between host physical address (HPA) and guest physical memory, the *tlbwe* in the guest machine must conduct a trap to ask for the hypervisor's help. The hypervisor searches the shadow TLB for the GVN to GPN mapping, and translates the GPN to HPN before inserting the $GVN \rightarrow HPN$ mapping into the hardware TLB. This trap and the hypervisor intervention introduce more than 1000 instructions overhead for the TLB miss.

To eliminate the overhead caused by *tlbwe*, a hardware extension is introduced in Power embedded cores, which conducts the GPN to HPN translation automatically and transparently. A new address translation table is built in hardware, called LRAT (Logical to Real Address Translation, here Logical==GPA and Real==HPA), providing a

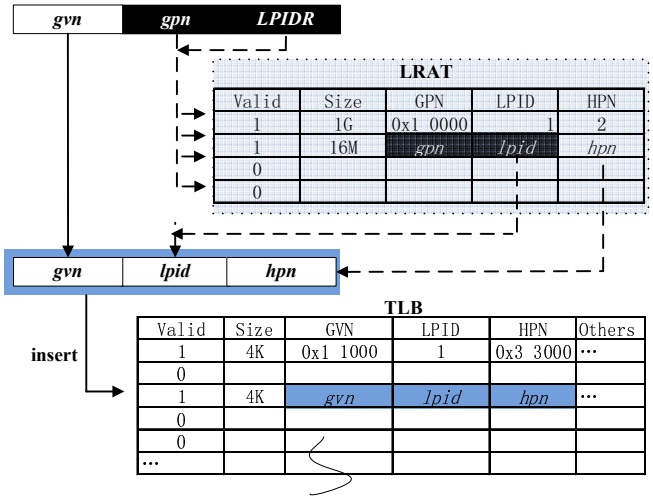


Figure 8: LRAT Usage for tlbwe Instruction in Guest Mode

fast lookup for GPA to HPA translation. The LRAT consists of multiple entries. Each entry should be assigned to a guest machine. On the other hand, a guest machine can have multiple entries. A high degree of flexibility is achieved through the support of different segment sizes in each entry. And it also has mechanisms to support memory holes in huge entry blocks, which can be used for small block sharing and fragmentation cases. The LRAT is owned and managed by the hypervisor.

With LRAT, writing TLB entries with *tlbwe* is supervisor privileged and is executable by either hypervisor or guest. When a *tlbwe* is executed in hypervisor, it works as before. When a *tlbwe* is executed in guest, the LRAT entries are searched for a matching GPN by the hardware. If one entry is found, the GPN is translated to an HPN before being written into the TLB. In the case where no match is found, an LRAT miss exception is raised that must be resolved by the hypervisor (e.g. allocate memory from the system and create an LRAT entry for the guest machine). The *tlbwe* instruction's workflow is shown in Figure 8.

Finally, combining all introduced mechanisms allows TLB miss handling to be completely executed in the guest machine without hypervisor intervention.

4. VIRTUALIZATION IMPLEMENTATIONS ON POWER EMBEDDED CORE

In this section, we describe our KVM-based hypervisor implementations with the hardware assisted extensions on the Power embedded A2 core.

4.1 IBM A2 Core

Based on the 64-bit Power ISA BookE[16] architecture, A2 cores support in-order execution and dual issue from up to four active hardware threads in a fine-grained simultaneous multi-threaded (SMT) fashion. Each A2 has a 16KB L1-I and L1-D cache and 512 TLB entries (4-way set associative). A group of four A2s share a 2MB L2 cache. A2 cores are currently deployed in two different systems, the IBM PowerEN processor[10, 11] and the IBM Blue Gene/Q[14]. The IBM PowerEN processor was designed to meet the demands

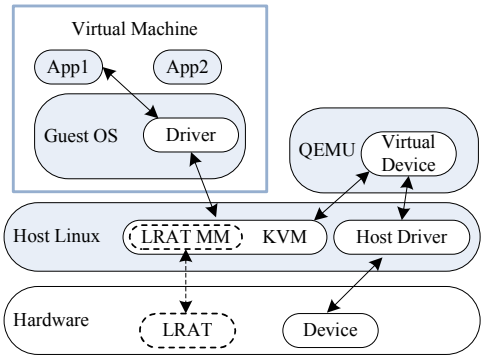


Figure 9: KVM Virtualization Infrastructure

of next generation network applications. It integrates 16 general purpose, 4-way SMT A2 cores, with an array of on-chip accelerators. BG/Q is the third generation computer architecture in the Blue Gene series[17] of supercomputer. Each BG/Q compute chip has 16 A2 cores. For reference, all experiments were conducted on a PowerEN system.

The hardware extensions described in Section 3 are all implemented in the A2 core. These extensions are designed to be configurable by the hypervisor. For example, the guest mode extension can be enabled by setting the MSR[GS]. LRAT can be enabled or disabled by setting *tlbwe* instruction to supervisor privileged or hypervisor privileged. The hardware overhead for these extensions is negligible, nor do they introduce complex logic or require large memory. For example, the LRAT in each A2 core is a small 8-way fully associative CAM (Content Addressable Memory). Each LRAT entry in the A2 core supports multiple "chunk" sizes, from 1MB to 1TB, which can be dynamically configured by the hypervisor based on different requirements of applications and scenarios.

In the A2 core, the privileged registers are decoupled to supervisor privileged and hypervisor privileged, allowing the guest OS to access the supervisor ones directly. There are 127 non-user SPRs in the A2 core and 52 among them are supervisor privileged. These 52 supervisor privileged SPRs include many high-use registers, such as MSR and the interrupt handler related registers, which can be accessed natively in the guest OS.

4.2 Kernel-based Virtual Machine Implementations

KVM[18] is an open source hypervisor implementation that is integrated into the Linux kernel and provides feature-rich and highly efficient virtualization. It consists of a loadable kernel module which provides the core virtualization infrastructure, and a processor specific module for the architecture emulation and interrupt handling. The KVM architecture is shown in Figure 9.

In order to evaluate each of the hardware extensions and determine their overall contribution to performance improvements, we enable them gradually. We implemented 3 different KVM hypervisors for the A2 core architecture. The implementations are based on the Linux 3.1 kernel. These versions are described in the following.

4.2.1 KVM for EMUL Virtualization:

The first hypervisor version is a pure software EMUL vir-

tualization implementation, which assumes the A2 core does not have any hardware assisted virtualization support. All hardware extensions in the A2 core are disabled at system start time. We leverage the KVM framework for Power embedded core and implement the A2 specific operations for EMUL virtualization. In order to model the hardware resources and emulate the instructions for the guest machine, several thousands lines of code are added to the hypervisor. The shadow TLB for each guest is modeled identical to the hardware TLB organization (e.g. number of TLB entries) in the A2 core.

Because in general a guest can not access I/O directly, I/O device virtualization in KVM is provided through QEMU[6]. For I/O accesses, a guest OS usually invokes a guest device driver, which then triggers an interrupt and switches execution into the KVM environment. KVM then transfers the device access request to QEMU for device emulation. If the access should be emulated by software, QEMU calls a software device model, emulates the corresponding device behaviors and returns the results. Otherwise, QEMU calls a host device driver to provide the desired I/O function. Its workflow is also shown in Figure 9.

4.2.2 KVM for GS Virtualization:

In the GS virtualization implementation, we enable all hardware extensions above, except MMU virtualization, i.e. LRAT and interrupt redirection are not enabled. The hypervisor still leverages the shadow TLB, and all TLB related instructions in the guest always cause a trap-and-emulate. The complexity of the hypervisor implementation has been reduced. For instance, due to the introduction of LPID, PID mapping in the hypervisor is not required anymore.

4.2.3 KVM for LRAT Virtualization:

Finally, we enable all hardware assisted virtualization features, and call this hypervisor version the LRAT virtualization. The shadow TLB is no longer needed, as the guest can directly access the hardware TLB, and $GPN \rightarrow HPN$ translation is handled transparently by the LRAT. However, the LRAT management module is now integrated with the hypervisor, which is shown as 'LRAT MM' in Figure 9.

Generally I/O accesses from the guest machine are always trapped to the hypervisor by the DTLB miss and then emulated by leveraging QEMU. However, in LRAT virtualization all guest TLB misses are redirected to the guest handlers, instead of trapping into the hypervisor. When the guest handler processes the DTLB miss for I/O accesses, it executes a *tlbwe* instruction, which causes the LRAT miss exception because the hypervisor has not inserted the address mapping for the I/O address. That LRAT miss exception traps to the hypervisor and the hypervisor can detect using the GPN with *tlbwe* that the guest OS is handling an exception caused by an I/O access. However, the specific address of the I/O access can no longer be detected because two exceptions (a DTLB miss and an LRAT miss) have occurred. Without the address, emulation for the I/O device can not be conducted.

One approach for guest I/O accesses is source-level modifications using a technique called paravitualization[5]. The source code of the guest OS is modified to create an interface that proactively invokes the hypervisor for the device virtualization. Paravirtualization offers high performance, but its one disadvantage is that it requires changes on the guest OS.

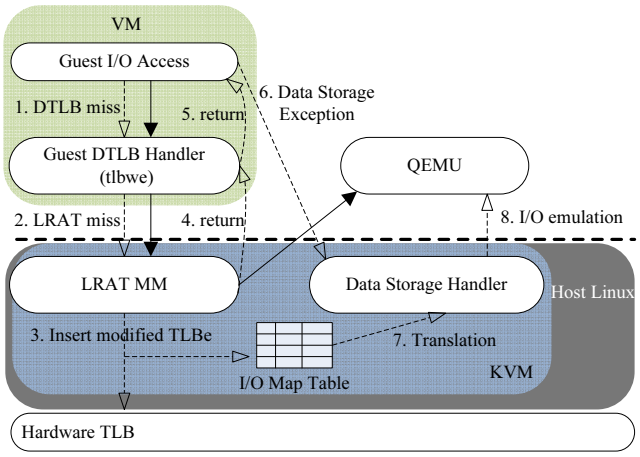


Figure 10: Guest I/O Device Access in LRAT Virtualization

A general choice is to leverage LRAT miss exception and data storage exception to conduct the I/O emulation. As shown in Figure 4, DTLB misses occur significantly more frequent than data storage exceptions. Hence redirecting storage exceptions back to the hypervisor will not introduce significant overhead. When an I/O access is detected during LRAT miss exception, LRAT MM inserts a hardware TLB entry which permission has been changed (for example, no read, no write or no execute) to cause a storage exception on the I/O pages (GPN). Meanwhile, LRAT MM maintains the GVN to GPN mapping in an I/O map table. When the guest tries to access the I/O device the second time with GVA, it causes a data storage exception rather than a DTLB miss exception. Since the data storage exception is configured by the hypervisor, the storage exception handler can detect the I/O access by the GVA from the guest context, and translate it to the GPA by the I/O map table. Then QEMU can be invoked to emulate the I/O access as it does before. The work flow of this guest I/O access approach takes eight steps and is shown as the dashed lines in Figure 10.

Our goal for LRAT virtualization is to simplify the implementation of a robust hypervisor, which can support a broad range of unmodified guest OSs, and to maintain high level of performance. Based on the approach above, we leverage the register mapping mechanism of the A2 core to build a simpler solution. For example, The Data Exception Address Register (DEAR) in the Power embedded core keeps the exception-causing address for Data TLB Error and Data Storage interrupts. When running in guest state, the DEAR will be mapped to the GDEAR. The specific address of the I/O access is reserved in the GDEAR during the guest TLB miss handler. When a *tlbwe* LRAT miss exception occurs, the new exception address (GPN) is kept in the DEAR. Hence, the LRAT MM can directly invoke QEMU to emulate the I/O device with the information in the guest mapping register. This is accomplished with only three steps and shown using solid line in Figure 10.

5. PERFORMANCE EVALUATION

This section evaluates the performance of the different virtualization mechanisms based on the Linux kernel 3.1

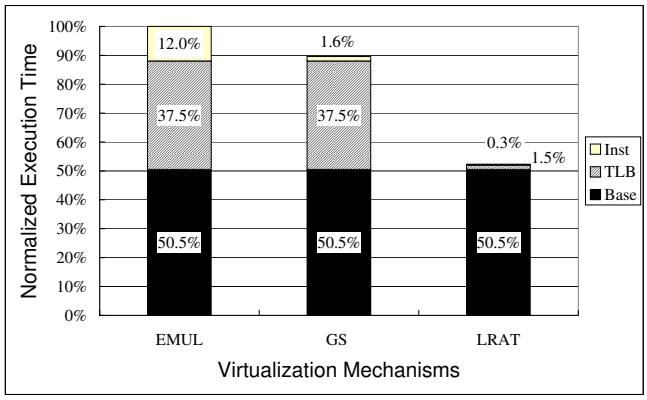


Figure 11: Performance Breakdown for Linux Booting

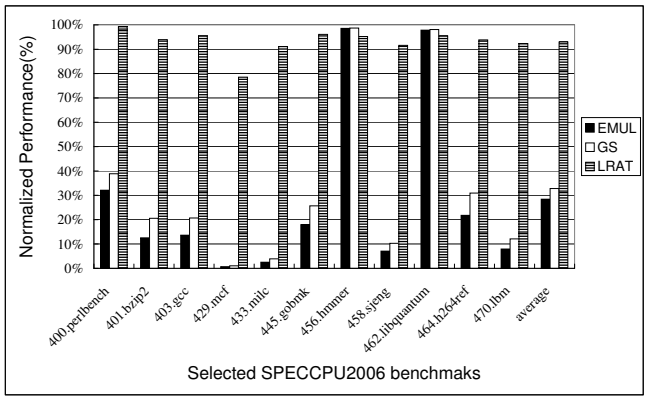


Figure 12: Performace of Virtualization for SPEC-CPU2006 Relative to Native Execution

booting process and the SPECCPU2006 benchmarks. The three KVM hypervisor versions are evaluated on the same IBM PowerEN platform with the Linux kernel 3.1 running as the host OS. The main memory is 16GB on the host PowerEN system, and the virtual machines are always configured to have 1GB memory in the following evaluations. Since Linux is widely used in virtual machine solutions, we choose guest Linux kernel booting process as a practical workload for evaluation, which has a complicated code path and various hypervisor mode switches. We measure the elapsed time from the first instruction until the init process starts running. SPECCPU2006 C test set is used as another computation-intensive workload to evaluate the virtualization performance. We have chosen to measure SPEC-CPU2006 over SPECvirt_sc2010 [24] as we want a finer grain analysis of the overhead that occurred, similar to [2].

5.1 Efficiency of Virtualization Mechanisms

Figure 11 shows the overhead breakdown of guest Linux kernel booting in different virtualization mechanisms. The GS virtualization mechanism reduces privileged instruction emulation overhead (from 12% to 1.6%), but fails to improve the performance of TLB misses, which is a major part of the virtualization overhead (37.5%). The LRAT mechanism further reduces the instruction emulation overhead to 0.3%, and the TLB miss overhead to less than 1.5%. The remaining 1.5% TLB miss overhead is due to MMIO exceptions, which can not be avoided and must be handled by QEMU.

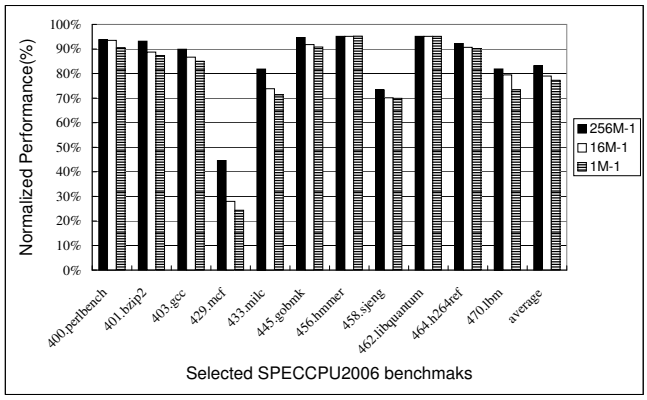
We also evaluate the efficiencies of different virtualization mechanisms by comparing the execution time of SPEC-CPU2006 benchmarks. The relative results of the SPEC-CPU2006 benchmarks running under 3 different virtualization mechanisms are shown in Figure 12. On average the efficiency of software virtualization (EMUL) is less than 30% of that of the native system, with GS virtualization having to some extent obvious improvements (+17.9%). However, with LRAT virtualization it achieves 93% of the performance of the native runs. That is comparable to hardware managed TLB mechanisms. We conducted the same experiments on x86 with KVM and SPECCPU2006, which also achieved about 93% performance of native runs. The privileged instruction optimization of GS virtualization has limited impact on performance. This might be due to SPECCPU2006 benchmarks being computation intensive applications with few privileged operations. However the TLB miss optimization in LRAT virtualization improves the performance significantly (on average 232%), since most of the virtualization overhead comes from TLB operations. Mcf benchmark in LRAT virtualization even gets 146X times performance gain than in EMUL virtualization. That is due to mcf being a memory-intensive program, which causes much more TLB misses exceptions than others do[15]. Its performance in EMUL virtualization is only 0.7% of native, which is not acceptable.

5.2 LRAT Organization Evaluation

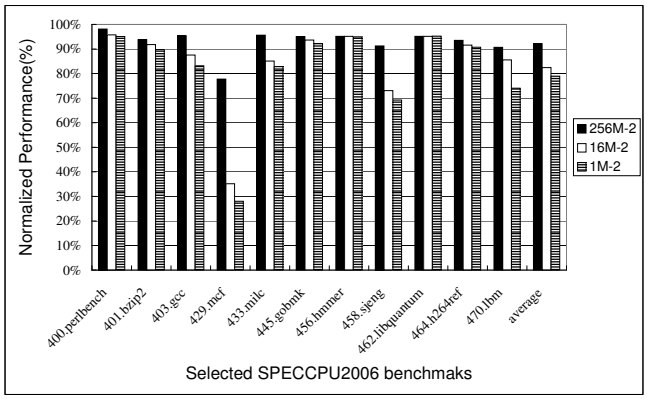
The evaluation reveals that LRAT is an important and effective component of hardware extensions for Power embedded core virtualization. One possible concern is whether the small LRAT table (8 entries) of the A2 core restricts the virtualization performance or not. In Figure 13a-d, we show the performance impact of different LRAT organizations relative to native execution. We evaluated different numbers of LRAT entries and different memory chunk sizes of each entry. In general, the benchmarks of SPECCPU2006 show good locality in data accesses. Thus 1M x 4 (4 entries with 1MB chunk size) or 16M x 2 for a guest is sufficient to keep the efficiency above 80% of native. The abnormal cases are mcf and sjeng, which access data randomly and with a wide stride. The experiments indicate that 256M x 2 for a guest machine is a reasonable choice. Adding more entries for a guest machine does not improve their efficiency. An 8-entry LRAT can support at least two guest machines per core without significant performance loss. The wide stride of entry chunk size selection can help to improve the performance behavior on abnormal cases like mcf and sjeng. In addition, a dynamic LRAT chunk allocation mechanism in the hypervisor based on TLB miss behavior analysis might further help to improve guest efficiency.

6. RELATED WORK

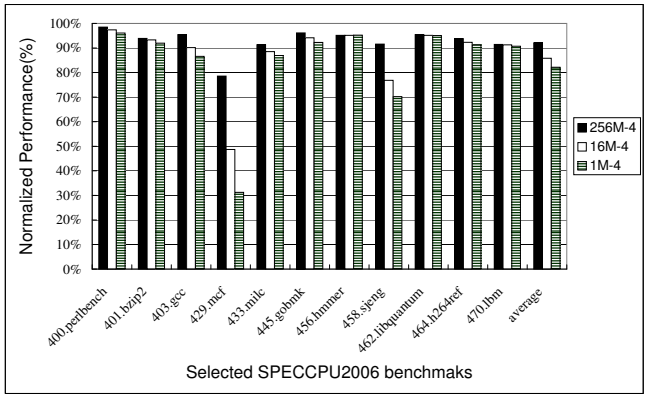
Virtualization is now found in almost all processors. Since virtualization introduces new overhead that origins from the virtualization of the CPU, the MMU and the I/O, many processor vendors have introduced hardware extensions to reduce the overhead. In this section, we briefly discuss the virtualization capabilities of the MIPS, UltraSPARC, X86 and ARM architectures.



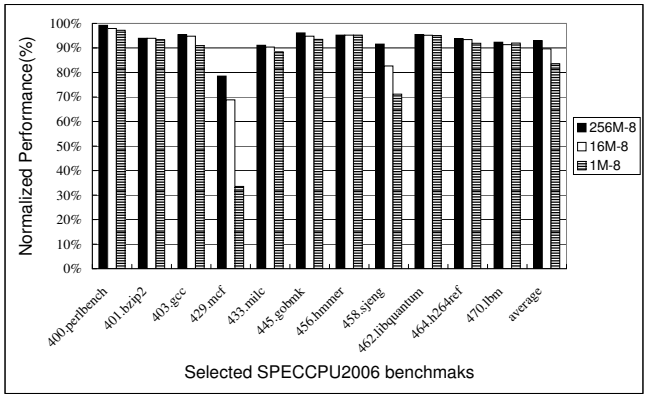
(a) 8 LRAT entries



(b) 4 LRAT entries



(c) 2 LRAT entries



(d) 1 LRAT entry

Figure 13: Performance Impact of Varying Numbers of LRAT Entries

MIPS and UltraSPARC also use software managed TLBs. To the best of our knowledge, there is no official hardware virtualization support in the MIPS architecture[20]. Hence, MIPS processors have to utilize shadow TLBs and the trap-and-emulate approach to virtualize their software managed TLBs. One can expect similar overheads as demonstrated in EMUL evaluation. UltraSPARC supports hardware assisted virtualization[25, 26]. However, the hypervisor software handles the translation from GPA to HPA. ARM has recently added full virtualization support as an IP option and has included it in their latest high end processor ARM Cortex-A15 MPcore[4].

X86 based architectures (Intel and AMD) use a hardware managed TLB. The hardware determines the $GVA \rightarrow HPA$ translation through a duality of page table translations and enters them into the TLB. Over the years both Intel and AMD have introduced hardware features to assist and improve virtualization. Both AMD and Intel introduced their first-generation support[21] for x86 virtualization (AMD-V and Intel VT-x technologies) but VMware demonstrated limited performance gains[1]. In their second generation of hardware support that incorporates MMU virtualization (Intel's EPT and AMD's RVI), performance was improved significantly (42%∼600%)[28, 27]. When EPT is active, the ordinary page tables (referenced by control register CR3) translate from linear addresses to guest-physical addresses. A separate set of page tables (the EPT tables or nested page tables) translate from GPAs to the HPAs that are used to access memory. As a result, guest operating systems can manage their own page tables and directly handle page faults[13].

Bhargava et al. discussed page walk overheads in virtualized systems and showed that a small number of entries for nested page tables are frequently reused due to spatial and temporal localities. They extended their architecture with page walk caches in non-virtualized systems to cover nested page walks[7]. In academia, Ahn et al proposed flat nested page tables with minor hardware extensions to reduce memory references required for 2D page walks[2]. They also proposed speculative inverted shadow paging to reduce the cost of a nested page walk to a single memory reference, without hypervisor interventions for guest page table changes. They evaluated their two schemes, which can improve the page walker 7% and 14% respectively.

7. CONCLUSION

In this paper, we discussed the design of hardware extensions to support virtualization in systems with software managed TLBs. These extensions were implemented in a Power embedded A2 core and evaluated on a port of an embedded KVM hypervisor. The extension focused on elimination of privileged instructions traps, introducing a TLB management and interrupt steering to the guest that do not lead to hypervisor traps thus significantly improving performance. A performance comparison with a pure soft-

ware implementation based KVM is conducted and we show that performance improvements are on average 232% better. With the support of hardware extensions, virtualization achieves 93% performance of native runs. The techniques introduced and evaluated in this paper are universally applicable to improving virtualization on architectures with software managed TLBs.

8. ACKNOWLEDGMENTS

This work benefited from the contributions of the whole PowerEN team at IBM: in particular, we thank Shahrokh Daijavad who drove and coordinated world-wide teams on KVM on PowerEN project. We also thank Tao Jiang for his help on evaluation with SPECCPU2006 and our anonymous reviewers for their invaluable comments.

9. REFERENCES

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on architectural support for programming languages and operating systems (ASPLOS)*, 2006.

[2] J. Ahn, S. Jin, and J. Huh. Revisiting hardware-assisted page walks for virtualized systems. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, 2012.

[3] AMD. *Virtualizing Server Workloads.* http://www.amd.com/jp/Documents/AMD_WP_Virtualizing _Server_Workloads-PID.pdf.

[4] ARM Architecture Group. *ARM Cortex-A15 Technical Reference Manuals*, Jan. 2011.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on operating systems principles (SOSP)*. ACM, 2003.

[6] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005.

[7] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on architectural support for programming languages and operating systems (ASPLOS)*, 2008.

[8] H. Blanchard. *KVM for Embedded PowerPC.* http://www.linux-kvm.org/wiki/images/c/cf/Kvm Forum2007%24KVM_Forum_-_Embedded_PowerPC.pdf.

[9] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo: a full system simulator for the powerpc architecture. In *ACM SIGMETRICS performance evaluation review*, 2004.

[10] J. Brown, S. Woodward, B. Bass, and C. Johnson. IBM Power Edge of Network processor: A wire-speed system on a chip. *Micro, IEEE*, 31(2):76–85, 2011.

[11] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development*, 54(1):3:1–3:11, 2010.

[12] Freescale. *e500mc Core Reference Manual*, Mar. 2012. http://cache.freescale.com/files/32bit/doc/ref_manual/E500MCRM.pdf.

[13] M. Gillespie. Best Practices for Paravirtualization Enhancements from Intel Virtualization Technology: EPT and VT-d. 2009. http://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization-technology-ept-and-vt-d.

[14] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q Compute Chip. *Micro, IEEE*, 32(2):48–60, 2012.

[15] J. L. Henning. Performance counters and development of SPEC CPU2006. *SIGARCH Computer Architecture News*, 35(1), Mar. 2007.

[16] IBM. *Power ISA Version 2.06 Revision B*, 2010. https://www.power.org/documentation/power-isa-version-2-06-revision-b/.

[17] IBM Blue Gene Team. Overview of the IBM Blue Gene/P Project. *IBM Journal of Research and Development*, 52(1/2):199–220, Jan. 2008.

[18] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, 2007.

[19] H. McGhan. SPEC CPU2006 benchmark suite. *Microprocessor Report*, Oct. 2006.

[20] MIPS Technologies. *MIPS Architecture.* http://www.mips.com/products/product-materials/processor/mips-architecture/.

[21] G. Neiger, A. Santoni, F. Leung, D. Rodgers, , and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, 2006.

[22] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[23] Standard Performance Evaluation Corporation. *SPECCPU2006.* http://www.spec.org/cpu2006/.

[24] Standard Performance Evaluation Corporation. *SPECvirt_sc2010.* http://www.spec.org/virt_sc2010/.

[25] Sun Microsystems. *UltraSPARC Architecture*, June 2006. http://opensparc-t1.sunsource.net/specs/UA2005-current-draft-HP-EXT.pdf.

[26] Sun Microsystems. *UltraSPARC Virtual Machine Specification (The sun4v architecture and Hypervisor API specification)*, Jan. 2006. http://opensparc-t1.sunsource.net/specs/Hypervisor-api-current-draft.pdf.

[27] VMware. *Performance evaluation of AMD RVI hardware assist*, 2009. http://www.vmware.com/pdf/RVI_performance.pdf.

[28] VMware. *Performance evaluation of Intel EPT hardware assist*, 2009. http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf.