The Evolution of an x86 Virtual Machine Monitor

Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, Pratap Subrahmanyam
{agesen, alextg, jeffshel, pratap}@vmware.com

ABSTRACT

Twelve years have passed since VMware engineers first virtualized the x86 architecture. This technological breakthrough kicked off a transformation of an entire industry, and virtualization is now (once again) a thriving business with a wide range of solutions being deployed, developed and proposed. But at the base of it all, the fundamental quest is still the same: running virtual machines as well as we possibly can on top of a virtual machine monitor.

We review how the x86 architecture was originally virtualized in the days of the Pentium II (1998), and follow the evolution of the virtual machine monitor forward through the introduction of virtual SMP, 64 bit (x64), and hardware support for virtualization to finish with a contemporary challenge, nested virtualization.

Categories and Subject Descriptors

C.0 [**General**]: Hardware/software interface; C.4 [**Performance of systems**]: Performance attributes; D.4.7 [**Operating Systems**]: Organization and design

General Terms

Performance, Design

Keywords

Virtualization, Virtual Machine Monitor, Dynamic Binary Translation, x86, vSMP, VT, SVM, MMU, TLB, Nested Paging.

1. INTRODUCTION

In 1970, engineers at Intel designed the world's first integrated microprocessor, the 4 bit 4004, with 2124 transistors running at 100 kHz [20]. 40 years later, over classics such as the 8008, 8080, 8088, 8086, 80286, 80386, 80486, and Pentium, this 4 bit architecture has evolved into a powerful 64 bit multicore CISC architecture ("x86"), with implementations containing 1+ billion transistors running at 3+ GHz. Amazingly, after this fantastic evolution, modern x86 processors still retain the ability to run ancient code directly.

As the x86 architecture evolved, it came to support a vast diversity of operating systems and applications. Its use expanded into almost every segment of the computing industry, from embedded systems and handheld devices to personal computers and servers, and from information processing to web services to high performance computing.

While trap-and-emulate virtualization on mainframes was well understood at the time, it was – unsurprisingly – not a design goal for the 4004, whose first application was in fact a Busicom calculator [7]. However, as x86 CPUs expanded their reach, the case for virtualization gradually emerged. There was just one problem, seemingly fundamental: the generally accepted wisdom held that x86 virtualization was impossible, or at best merely impractical, due to inherited architectural quirks and a requirement to retain compatibility with legacy code.

This impasse was broken twelve years ago when VMware engineers first virtualized the x86 architecture entirely in software. By basing the virtual machine monitor (VMM) on binary translation, the architectural features that prevented trap-and-emulate virtualization of the x86 architecture could be handled precisely and efficiently [1]. This technological breakthrough kicked off a transformation of an entire industry: five years ago, Intel started shipping CPUs with hardware support for instruction set virtualization (VT-x), three years ago AMD started shipping CPUs with hardware support for memory virtualization (RVI), and virtualization is now one of the most important ways to utilize the increasing number of cores per socket delivered by the continued run of Moore's law. Today, virtualization provides server consolidation, fault tolerance, security and resource management, and facilitates test, development and deployment across a multitude of operating systems and versions. Moreover, virtualization is the foundation of the ongoing shift towards cloud computing.

Still, and common to all these use cases, the fundamental quest remains the same: running virtual machines (VMs) as well as we possibly can on top of a VMM.

Previous publications have described specific technical aspects of VMware's x86 VMM, including device virtualization [17], binary translation [2], and a comparison between software and hardware approaches [1]. The present paper takes a historical approach, looking back over the past decade in approximate chronological order. We first distinguish between hypervisor and VMM to establish a context for the remainder of the paper (Section 2). Then we go back to 1998 to review how the x86 architecture was virtualized using binary translation in the Pentium II era (Section 3). Our next stop is 2003, where we introduce virtual SMP (Section 4). By 2004 the new feature is x64, AMD's 64 bit extension of the x86 architecture (Section 5). A year later, in 2005, hardware support for virtualization emerged with further support arriving in 2007-8 (Section 6). Finally, we review a contemporary challenge, nested virtualization (Section 7), briefly list other topics that space limitations do not permit us to discuss in detail (Section 8), and then offer our concluding remarks (Section 9).

2. CONTEXT

Before describing details of the x86 VMM, let us differentiate between hypervisor and VMM. Existing literature mostly treats these terms as synonyms, but for the purposes of this paper it is beneficial to ascribe a narrower role to the VMM. In our vocabulary, a *VMM* is an entity specifically responsible for virtualizing a given architecture, including the instruction set, memory, interrupts, and basic I/O operations. A *hypervisor* combines an operating system (OS) with a VMM. The OS may be a general-purpose one, such as Linux or Windows, or it may be one developed specifically for the purpose of running VMs.

For example, VMware's vSphere ESX hypervisor is comprised of the *vmkernel* and a VMM. The vmkernel contains a boot loader, an x86 hardware abstraction layer, I/O stacks for storage and networking, as well as memory and CPU schedulers. To run a VM, the vmkernel loads the VMM, which encapsulates the details of virtualizing the x86 architecture, including all 16 and 32 bit legacy modes as well as 64 bit long mode. The VM executes directly on top of the VMM, touching the hypervisor only through the VMM surface area.

VMware's products create a separate instance of the VMM for each running VM ("guest") rather than using a single VMM to run all VMs on the "host"; see Figure 1. This approach simplifies the overall system by providing a clear separation of concerns: the vmkernel manages host-wide tasks such as scheduling, while VMMs manage per-VM tasks. Moreover, it leads to a natural separation of mechanism, mostly implemented in the VMM, and policy, mostly implemented in the vmkernel, using knowledge of all the VMs running on the host. Finally, with multi-tenancy, where separate customers buy VM execution time on the same host, the use of a separate VMM for each VM greatly simplifies reasoning about isolation, although it does not enforce isolation as the VMM is a privileged program. (Resource management, and quality of service properties still remain to be satisfied globally by the vmkernel.)

While we have described this architecture in the context of VMware's ESX hypervisor, the same general idea extends to our hosted products such as VMware Workstation and Fusion: here, the VMM runs alongside the host OS, again with one VMM instance per virtual machine. In fact, other than a "shim" platform layer, the same VMM is used in all of VMware's products.

3. VIRTUALIZING 32-BIT x86

In 1998, the x86 architecture had no hardware support for virtualization and was generally considered to be unvirtualizable [15]. At that time virtual machines were typically run using an approach known as trap-and-emulate [13, 14]. In a trap-and-emulate style VMM, the guest code runs directly on the CPU, but with reduced privilege. When the guest attempts to read or modify privileged state, the processor generates a trap that transfers control to the VMM. The VMM then emulates the instruction using an interpreter and resumes direct execution of the guest at the next instruction.

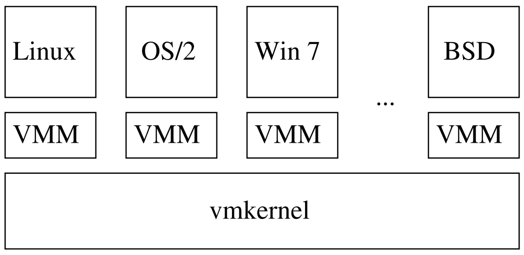

Figure 1: The ESX hypervisor: one vmkernel per host, and one VMM per virtual machine.

The x86 architecture was considered non-virtualizable because the processor could not be configured to generate the required traps[1]. One example is the `popf` instruction which loads a set of flags from the stack into the `%eflags` register. When executed in privileged mode, `popf` loads all flags, which include a mix of ALU flags (zero flag, carry flag, etc.) and system flags (interrupt flag, I/O privilege level, etc.). When executed in user mode, system flags remain unmodified regardless of the contents of the stack. Instructions like `popf` thwart any attempt to build a trap-and-emulate VMM for x86. If a deprivileged guest kernel attempts to clear the interrupt flag using `popf`, no trap is generated and the VMM has no way of knowing it should not deliver interrupts to the guest.

One could certainly avoid the reliance on traps by running the guest on an interpreter, but this introduces too much overhead to allow virtual machines to be used widely. Binary translation (BT), however, offers the ability to intercept any guest instruction and modify its execution with just as much flexibility as full interpretation, but with much lower overheads. To further improve performance, BT can be limited to guest kernel mode code. Guest user mode code can run directly on the CPU in "direct execution," as would be done in a trap-and-emulate VMM, since it already expects to run with user mode semantics such as ignoring attempted changes to the interrupt flag. Combining BT and direct execution limits translator overheads to the time the guest spends running kernel code, which is typically a minority of total execution time.

The following sections provide a more detailed description of how a BT-based VMM runs a guest. Section 3.1 describes a simple, yet highly efficient dynamic translator for unrestricted x86 code. Sections 3.2 and 3.3 show how to use paging and segmentation to establish an environment in which the translated code can execute. Finally, Section 3.4 outlines an adaptive feedback-based optimization that cus-

[1]The x86 architecture distinguishes between *faults*, which abort execution of an instruction prior to committing any state change, and *traps*, which happen after an instruction completes. Technically, an x86 VMM would need to use faults. However, for consistency with classical literature, we prefer the term "trap-and-emulate," even though the precise x86 term should be "fault-and-emulate."

tomizes translations to observed guest behavior.

3.1 Binary Translation

To build a VMM that can run every possible guest requires a binary translator that can handle the entire x86 instruction set. The translator provides the ability to intercept precisely the virtualization sensitive instructions without requiring trap semantics. Since we cannot patch instructions in the guest's own image (such patching would be visible to the guest), we must translate all guest kernel code and always execute from a guest-invisible translation cache. Fortunately, our input and output are both x86, so the instructions that do not need interception can be translated with little or no modification.

Our translator has these properties:

- *Binary.* Input is binary x86 code, not source code.
- *Dynamic.* Translation happens at runtime, interleaved with execution of the generated code.
- *On demand.* Code is translated only when it is about to execute. This laziness side-steps the problem of distinguishing code from data.
- *System level.* The translator makes no assumptions about the guest code. Rules are set by the x86 ISA, not by a higher-level ABI. In contrast, an application-level translator like Dynamo [4] might assume that "return addresses are always produced by calls" to generate faster code. The VMM does not: it must run a buffer overflow that clobbers a return address precisely as it would have run natively (producing the same hexadecimal numbers in the resulting error message).
- *Subsetting.* The translator's input is the full x86 instruction set, including all privileged instructions; output is a subset (mostly user-mode instructions).

To explain how the translator operates, we work through a short example, a spin lock function that acquires a lock and returns to the caller with interrupts disabled. (Such "irq locks" may protect kernel data structures that are accessed both from normal and interrupt handling contexts.)

```
void SP_LockIRQ(SPLock *lock) {
  DisableInterrupts();
  while (CompareExchange(lock, 0, 1) != 0) {
    EnableInterrupts();
    Pause();
    DisableInterrupts();
  }
}
```

We compiled the C code into this 32-bit binary:

```
        push  %ebx                ; callee saved
        mov   %eax,%edx           ; %edx = %eax = lock
        cli                       ; disable interrupts
        mov   $1,%ecx             ; %ecx = 1
        xor   %ebx,%ebx           ; %ebx = 0
        jmp   doTest
spin:   sti                       ; enable interrupts
        pause                     ; yield hardware thread
        cli                       ; disable interrupts
doTest: mov   %ebx,%eax
        lock                      ; If %eax==(%edx) write
        cmpxchg %eax,%ecx,(%edx); %(edx) = %ecx else
                                  ; %eax = (%edx)
        test  %eax,%eax           ; Set flags from %eax
        jnz   spin                ; Jump if not zero
done:   pop   %ebx
        ret
```

We invoked this code in a virtual machine, logging all code translated. The above code is not the input to the translator; rather, its binary ("hex") representation is input:

```
53 89 c2 fa b9 01 00 00 00 31 db ...
```

The translator reads the guest's memory at the address indicated by the guest program counter (`%eip`), classifying the bytes as prefixes, opcodes or operands to produce intermediate representation (IR) objects. Each IR object represents one guest instruction.

The translator accumulates IR objects into a translation unit (TU), stopping at 12 instructions or a terminating instruction (usually control flow). The fixed-size cap allows stack allocation of all data structures without risking overflow; in practice the cap is rarely reached since control flow tends to terminate TUs sooner. Thus, in the common case a TU is a basic block. The first TU in our example is:

```
push  %ebx
mov   %eax,%edx
cli
mov   $1,%ecx
xor   %ebx,%ebx
jmp   doTest
```

Translating from x86 to x86 subset, most code can be translated IDENT (for "identically"). The `push`, `mov`s, and `xor` all fall in this category.

Since `cli` is a privileged instruction, it must be handled specially by the VMM. Unlike `popf`, `cli` can generate a trap, but it is more efficient to avoid the trap by translating it non-identically. The `cli` translation must clear the virtual interrupt flag, which exists as part of an in-memory image of the guest's virtual CPU (VCPU), so we use an `and` instruction:

```
and   $0xfd,%gs:vcpu.flags
```

Since the translator does not preserve code layout, all control flow instructions must use non-IDENT translations. In this example the `jmp` becomes a translator-invoking continuation (indicated by square brackets). Putting this together, the resulting translation from the first TU of our example is:

```
push  %ebx
mov   %eax,%edx
and   $0xfd,%gs:vcpu.flags
mov   $1,%ecx
xor   %ebx,%ebx
jmp   [doTest]
```

Each translator invocation consumes one TU and produces one compiled code fragment (CCF). Although we show CCFs in textual form with labels like `vcpu.flags`, in reality the translator produces binary code directly.

After producing the above CCF, the VMM will execute the code which ends with a call to the translator to produce the translation for `doTest`. This second TU is all IDENT except for the final conditional `jnz` branch for which the translator emits two continuations (one for each successor):

```
jnz   [spin]
jmp   [done]
```

To speed up inter-CCF transfers, our translator, like previous ones [9], employs a "chaining" optimization, allowing one CCF to jump directly to another without calling out of the translation cache (TC). These chaining jumps replace the continuation jumps, which therefore are "execute once." Moreover, it is often possible to elide chaining jumps and fall through from one CCF into the next.

For conditional branches, at most one of the two successors can use fall through. The other must remain in the translated code as a conditional branch, initially invoking the continuation, but, once the translated target is produced, redirected to this target. (Sometimes, to avoid code duplication, no successor can use fall-through, so the final translation uses a `jcc`/`jmp` pair of instructions to connect to each of the successors.) Since translation and execution interleave, the first of the two continuations to execute is most likely to receive the beneficial fall-through treatment. If the first and subsequent executions follow similar paths, this tends to straighten code for good i-cache performance. In effect, the translator builds execution traces in the TC, even as it works through guest code in smaller TU chunks.

This interleaving of translation and execution continues for as long as the guest runs kernel code, with a decreasing proportion of translation as the TC gradually captures the guest's working set. For the spin lock example the translation, after one spin-free acquisition, results in this code in the TC:

```
* push    %ebx                   ; IDENT
  mov     %eax,%edx
  and     $0xfd,%gs:vcpu.flags   ; PRIV
  mov     $1,%ecx                ; IDENT
  xor     %ebx,%ebx
* mov     %ebx,%eax              ; IDENT
  lock
  cmpxchg %eax,%ecx,(%edx)
  test    %eax,%eax
  jnz     [spin]                 ; JCC
* pop     %ebx                   ; IDENT
* mov     %eax,%gs:scratchEAX    ; RET_LAUNCH
  mov     %ecx,%gs:scratchECX
  pop     %eax
  movzx   %al,%ecx
  jmp     %gs:rtc(4*%ecx)
```

Above, there are four CCFs with the leading instruction in each one marked with an asterisk. The continuation to the `spin` label remains untranslated as it has not executed yet. The code that was executed now sits in a straight line without jumping about as the original code did.

The last CCF above terminates with a "launch" sequence for a return translation, the details of which have been described previously [2].

For a bigger example than the spin lock, but nevertheless one that runs in exactly the same manner, booting Windows XP Professional and then immediately shutting it down translates 933,444 32-bit TUs and 28,339 16-bit TUs. While this may seem like a lot, translating each unit takes just 3 microseconds for a total translation time of about 3 seconds. Against a background of a one minute boot/halt, and keeping in mind that a boot workload has an unusually high proportion of cold code, the cost of running the translator is acceptable.

The translator does not attempt to "improve" the translated code. We assume that if guest code is performance critical, the OS developers have optimized it and a simple binary translator would find few remaining opportunities. Thus, instead of applying deep analysis to support manipulation of guest code, we disturb it minimally.

Most virtual registers bind to their physical counterparts during execution of TC code to facilitate IDENT translation. One exception is the segment register `%gs`. It provides an escape into VMM-level data structures; see Section 3.3. The `ret` translation above uses `%gs` overrides to spill `%eax` and `%ecx` into VMM memory so that they can be used as working registers in the translation of `ret`. Later, of course, the guest's `%eax` and `%ecx` values must be reloaded into the hardware registers.

As with registers, the translator binds guest ALU-flags (`CF`, `PF`, `AF`, `ZF`, `SF`, `OF`) to their physical counterparts. Since many x86 ALU instructions modify flags, nontrivial translations often must save and restore guest flags around flags-clobbering operations. For example, this applies to the `cli` translation, where the use of `and` clobbers guest flags. However, in the above example, the translator avoided flags save/restore code by looking ahead to see that the guest soon will execute an `xor`, which (re)defines all flags. To ensure that even the guest's interrupt handler has a consistent view of flags, the VMM defers virtual interrupts until the `xor` that terminates the flags-optimized region.

3.2 Virtualized Memory: Shadow Page Tables

The x86 architecture has supported virtual memory since the 80386 with an MMU consisting of a TLB and a hardware page table walker. The walker fills the TLB by traversing hierarchical page tables in physical memory. Originally, these page tables were two levels deep but were extended to three and later four levels (see Section 5). The walker may speculate ahead and cache mappings whenever it pleases, and is allowed to keep those mappings in the TLB until software explicitly flushes the TLB entry with an `invlpg` instruction or by switching contexts (assigning to the page table root control register `%cr3`). Additionally, the TLB is allowed to evict entries unpredictably, at any time.

The VMM's MMU serves two purposes. It maintains isolation of the VMM by ensuring that the guest cannot access memory belonging to the hypervisor, the host, or another VM. It also ensures that the virtual address space visible to the guest reflects what the guest expressed via its in-memory page tables. These concerns can be viewed as two separate layers of indirection stacked on top of one another: (1) the guest creates page tables to describe how to map guest virtual addresses (gVAs) to guest physical addresses (gPAs), and (2) the VMM creates mappings from gPAs to host physical addresses (hPAs).

To provide native-speed memory access for direct execution and IDENT translations (see Section 3.1), the composite mapping from gVA to hPA ultimately must reside in the hardware TLB. This can be achieved by pointing `%cr3` at a "shadow page table" that stores the composite mappings.

The cost paid by the first access is, however, considerably more than a traditional TLB miss cost. To understand this, we must examine what happens when the guest accesses a particular gVA. First, the memory access causes a page fault (several hundred cycles in the circa 2002 processors). Then, the VMM walks the guest's page tables in software to determine the gPA backing that gVA (again costing a few hundred cycles). Next, the VMM determines the hPA that backs that gPA. Often, this step is fast, but upon first touch it requires the host OS to allocate a backing page. Finally, the VMM allocates a shadow page table for the mapping and wires it into the shadow page table tree. The page fault and the subsequent shadow page table update are analogous to a normal TLB fill in that they are invisible to the guest, so they have been called "hidden page faults." Hidden faults can have a 1000-fold increase in cost over a TLB fill, but tend to be less frequent due to higher virtual TLB capacity (i.e., higher shadow page table capacity).

Once the guest has established its working set in the shadow page table, memory accesses run at native speed until the guest switches to a different address space. TLB semantics on x86 require that context switches flush the TLB, so a naive MMU must throw away the shadow page table and start over. We say such an MMU is "noncaching."

The only way to avoid the cost of rebuilding the shadow page table after each address space switch is to keep around copies of multiple shadow page tables, one for each address space. Here, the x86 semantics of implicit TLB flushes on context switches gets in the way. If a guest changes a mapping in its page tables after a shadow entry has been created, then switches to a new address space, and then switches back to the first address space, the guest can expect the new mapping to be visible because of the implicit TLB flush.

To solve this problem, the MMU needs a way of detecting when the guest modifies a primary page table entry which has been used to create a shadow page table entry. This is solved with a general purpose mechanism in the VMM known as *traces*, which mediate access to guest memory by the guest or by the VMM. Traces provide clients with notification upon access to pages of interest. The MMU uses traces for shadow page table consistency, but it is also partially responsible for implementing traces. A traced access may be detected when it results in a page fault caused by an attempted access, whether by guest or VMM. The page-fault handler then notifies the trace clients before resuming execution. Alternatively, traces may be "fired" on memory-accesses through adaptive techniques like those described in Section 3.4, or through explicit checks in the VMM itself.

With traces, it is possible to avoid throwing away the shadow page table when switching to a new address space. Instead, the MMU can selectively invalidate shadow page table entries when they are modified. Even then, the cost of hidden faults is one of the top virtualization overheads.

To further reduce the number of hidden faults, the MMU can combine the trace processing and the (re)validation of the shadow page table into a single step, using a technique called "eager validate." When the MMU receives a notification that the guest has modified a page table, instead of invalidating the shadow, the MMU immediately populates it with an updated entry. Now the guest can access the memory addresses mapped by the page table, without incurring a hidden fault. Although physical CPUs generally would continue using the old mapping until the next TLB flush, they are allowed to evict the mapping and refetch from page tables any time. So a VCPU using eager validate appears to have a TLB that always drops and reloads entries immediately when the in-memory page table is modified.

Eager validate has its own costs since it requires receiving a notification whenever a page changes. Page table entries might be modified, but never used, or a piece of memory that was once a page table might be used for other purposes. Either way, there is a pressure on the MMU to remove the trace if it is not helping to reduce page faults.

This leads to a tension between three paths that all MMU tuning revolves around:

- The hidden page fault that occurs when the guest accesses a page not mapped by the shadow page table.
- Switching from one address space to another (on behalf of the guest) and making sure the new shadow page table space is in sync with the guest page table.
- Guest writes to page table entries that the VMM is monitoring with a trace.

The overheads of these three operations all trade against one another. For instance, it is possible to eliminate all costs when the guest modifies a page table by not tracing the page, but this incurs more hidden page faults. Alternatively, hidden faults can be eliminated by eagerly shadowing entire address spaces, but this increases costs to switch address spaces. Finally, it is easy to make context switches quick, but only when the shadow page table is in sync with the guest's page table.

Finding the right balance between the three is very difficult and varies from workload to workload.

In addition to this three-way cycle trade-off, the MMU must factor in shadow page table memory usage. Some workloads, like Windows Remote Desktop Services, have working sets so large that they overwhelm any reasonable fixed allocation for shadow page tables, causing shadow page table eviction, i.e., thrashing. To counter this problem, the MMU must optimize shadow eviction, applying an LRU-like reclamation strategy. Moreover, the MMU must dynamically size the shadow page table cache according to measured needs, balancing the memory consumption against the utility of other possible uses.

3.3 Segmentation

The guest expects to use the full feature set of the x86 architecture, including the entire virtual address space. This presents a challenge to the VMM, which must simultaneously satisfy the guest's expectation as well as provide for its own needs. In particular, the translator must keep the TC and auxiliary data structures mapped to permit execution of translated code. For example, when running a 32 bit guest on a 32 bit CPU, the guest and VMM must somehow share the 4 GB address space in a manner that is both efficient and invisible to the guest.

A common way of controlling accesses to memory is through the use of page permissions. Abstractly, the VMM may set up mappings in the shadow page tables to disallow guest access to the VMM portion of the address space. For example, the VMM could use supervisor-only mappings for the VMM address range and user mappings for the (user mode part of the) guest address range. This approach works well with a trap-and-emulate style VMM; it is also the approach we use when running guest user mode code in direct execution.

However, page-based separation requires that the guest and the VMM run at different privilege levels so it does not work well for running translated guest kernel code. When running translated code, the TC itself must be accessible (for execution), yet must remain invisible to guest data accesses. While execute-only mappings could accomplish this, x86 page tables have no support for execute-only mappings. In addition, an execute-only approach would not apply to VMM-defined auxiliary data structures that must be accessible to translated code yet inaccessible when referencing memory on behalf of the guest.

Having ruled out paging as a way to protect the VMM from guest memory references in translated code, which approaches remain? One possibility is to use the BT infrastructure itself to insert explicit bounds checks into the translation of guest instructions. However, using bounds checks on every guest memory access causes significant overhead.

Fortunately, the x86 architecture has one more mechanism that can be applied: segmentation. The segmentation support goes back to the 16 bit 80286 processor, predating even paging. Over the years, segmentation was carried forward into 32 bit processors, although modern operating systems have largely discontinued reliance on segmentation, preferring the additional flexibility offered by paging. Segmentation gives us almost exactly what we need to enforce the boundary between guest and VMM within a shared address space: a fast bounds check on memory accesses.

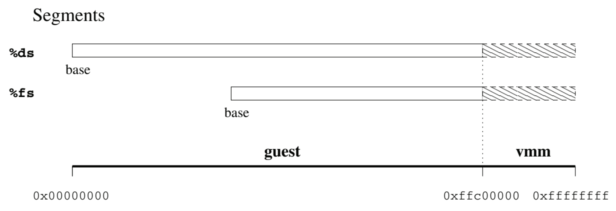

Figure 2: Segment-truncation protecting the upper 4 MB containing the VMM address-space.

In more detail, the x86 architecture provides six segment registers: one is used by instruction fetches (`%cs`), one for stack accesses (`%ss`), and the remaining four for other data accesses (`%ds`, `%es`, `%fs`, `%gs`). Each segment register contains a base and a limit and will generate a fault if used with an address greater than the limit. By configuring the data segments to allow access only to the guest portion of the address space, but allowing the code segment `%cs` to access the VMM space, we have achieved the code/data split that a BT-based VMM needs to hide the TC. The VMM can differentiate between guest data accesses and translator accesses to runtime data the same way by restricting `%ds`, `%es` and `%fs` to the guest range while reserving `%gs` for the VMM range; see Figure 2. When the translator emits instructions that should read or write VMM data, it includes a `%gs` prefix. When translating a guest memory access, the translator makes sure to *not* use `%gs`. If a guest instruction uses `%gs`, the translator must strip the prefix and replace it with another segment prefix, say `%fs` (setting up `%fs` according to what the guest expected to have in `%gs`). Alternatively, the translator can emit code to perform the virtual `%gs`-segmentation "in software" without relying on a physical segment register.

Using segmentation to isolate guest memory accesses from the VMM solves one problem, but the VMM must also emulate the guest's use of the same segmentation functionality. Most modern operating systems make little or no use of segmentation. They generally define their segments to have a base of zero and a maximal limit, a case we refer to as a "flat" segment. Since flat segments are common, we place the VMM at the top of the address space so that flat segments can be precisely "truncated" to prevent access to the VMM while allowing access to all remaining virtual addresses. Historically, VMware chose to allocate the top 4 MB of the virtual address space to the VMM. A non-flat guest segment may wrap around the top of the 4 GB address space. Truncation then must cut away not just the part of the segment that overlaps the VMM but also the wrap-around portion, preventing direct guest access to addresses that are not used by the VMM.

The use of segmentation to separate guest memory accesses from those introduced by the translator is a critical component of VMware's ability to implement a low overhead x86 to x86 translator. Guest memory accesses that do not overlap with the VMM incur no overhead relative to that same memory access running "natively" outside of a VM. The major difficulty with this design has been the performance cliff for instructions that do overlap with the VMM. Such instructions incur a fault and must be emulated without the benefit of the shadow page tables for getting to the corresponding guest memory. Fortunately, the VMM can often avoid such faults by predicting if an instruction may generate memory accesses in the VMM range (see Section 3.4), but even then the instruction will typically execute one or two orders of magnitude slower than a native memory access.

This cliff leads to a strong pressure to minimize the frequency that guest memory overlaps with the VMM's address space. Because only the limit of the segment can be adjusted, the VMM is constrained to the top of the address space. In practice, it has also been limited to use at most 4 MB of address space. Making it any larger is undesirable as the VMM would then overlap with hot data structures for many Windows guests. The resulting scarcity of address space has influenced the design of almost all components of the VMM. Small data structures were squeezed as tightly as possible and large data structures had to be paged to leave space for a sufficiently large TC to hold the guest's code working set, even as guests grew over the years from Windows 95/98, NT, 2000, XP, and Vista to Windows 7.

While the address space pressure at times was a source of grief, over the years it created an environment that favored minimality and cleanliness as overriding design principles. We believe that this, in turn, has given us a more reliable and more efficient VMM.

3.4 Adaptive BT

The need to cheaply interpose on certain non-virtualizable instructions motivated the use of BT for guest kernel code. But as the VMM matured, BT became increasingly useful as a means of reducing additional virtualization overheads.

Maintaining coherency of the shadow page tables requires trapping accesses to guest page tables, which is accomplished by marking those pages as read-only in the shadow page tables. Each time the guest writes to a page table natively it is a simple write to memory, but when run in a VM it can turn into a page fault, followed by emulation of the instruction, and finally an update to the shadow page table.

Fortunately operating systems typically modify page tables from within blocks of code with the express purpose of modifying page tables. Thus, instructions that modify page tables always (or usually) modify page tables and instructions that do other things (like updating scheduler records) never modify page tables. This permits the VMM to track instructions that modify page tables and feed that information back into the translator, allowing creation of custom translations for instructions that are likely to modify page tables.

Since these translations assume that the page they are writing to is a page table (i.e., write-protected by the VMM), there is no need to attempt to write to the page only to receive a page fault. Instead they can jump directly to the instruction interpreter, avoiding the page fault and thereby eliminating a large portion of the overhead.

It is possible to save more than just the page fault. Rather than using a general-purpose interpreter that must decode the guest instruction, the translator can pre-decode the instruction into an easy-to-interpret form and call an optimized interpreter.

Reducing page table modification costs has other indirect benefits: we reduce the pressure on the MMU to drop (and reinstall) traces, in turn avoiding hidden page faults.

Optimized translations for instructions that touch special memory are useful beyond page tables: they can avoid page faults for accesses to memory-mapped devices by identifying instructions likely to touch such devices. For certain device accesses, such as those to the APIC, the VMM goes one step further.

The APIC is a memory mapped device built into every modern x86 CPU to control the delivery of interrupts. One of the device registers in the APIC memory range is the Task Priority Register (TPR) which specifies the minimal priority for an interrupt to be delivered. Many versions of Windows, starting with NT and up to but not including 2003 SP2, access the TPR tens of thousands of times per second. Worse still, certain antivirus software can increase the access rate by an order of magnitude. For example, running a compilation job on Windows XP with 2 VCPUs and antivirus software generated more than 800,000 accesses to the TPR per second.

Incurring a page fault each time Windows accesses the TPR results in unacceptable performance. Even replacing the page fault with a call to the interpreter can be too costly, given this access frequency. However, by using adaptive BT, we can emit custom translations for instructions that consistently access the APIC TPR. In ideal circumstances, the translator can prove that an instruction will always access a fixed virtual address that maps to the APIC. By cooperating with the MMU to receive notification if that address were to be remapped (the APIC lives in the physical address space), the translator can emit highly optimized translations. In the case of a guest write to the TPR, the best case translation contains about half a dozen load/store and ALU instructions to verify the pending interrupt priority and store the new task priority value (in the common case, no interrupt is raised). On a modern superscalar CPU, we measured this translation at just 3 cycles, exceeding even the native performance of an APIC TPR write.

Not all guest APIC accesses can be statically proven to access a fixed location in memory; in the remaining cases, the translator inserts checks to verify that the accessed address matches the predicted one. If the prediction fails (which is extremely infrequent), we take a slow recovery path. If it fails repeatedly, we can retranslate, using this new information.

In addition to those for the APIC and the MMU, the translator also customizes accesses to segment descriptor pages, to certain virtual network adapters, and, as mentioned previously, to the VMM's address range.

4. SUPPORTING MULTIPROCESSOR VMs

In early 2002, VMware needed to grow beyond its base in the desktop and developer focused workstation market. To capture server workloads, we considered virtualizing a recognized server architecture such as Itanium or SPARC. Ultimately, we continued to bet on x86 and set out to implement virtual SMP to have a story for larger workloads.

This project required changes and innovations throughout the hypervisor: the vmkernel scheduler was challenged; the virtual BIOS had to change; we needed to come up with a threading model, implement synchronization primitives that would work across multiple contexts, find a new device programming model, generalize traces, understand cross-CPU code modification, etc. It was all new, unknown and uncertain; in short, it was a "bet the company" effort for our start-up.

Amazingly, by the end of that summer, we had managed to boot a 4-VCPU Windows 2000 VM, albeit over several hours and making good use of checkpoints along the way to recover from crashes. The VM ran, but barely: perfmon running in the guest showed four CPUs, but it was so slow that the CPUs were pegged at nearly 100% load simply servicing timer interrupts! In spite of this dismal performance, we were buoyed by the success, and confident that optimization would address the performance and scaling issues. With little hesitation, we sent a screenshot to everyone in the company trumpeting success.

From that early milestone it took a significant amount of work to get anywhere near production quality. We had to restructure much of the VMM. Most importantly, we reimplemented traces in a vSMP-aware manner as a foundation for a new caching MMU. We also used the new trace implementation to provide coherency of translated code, and to shadow the descriptor tables so that direct execution could be reenabled. Much of the performance tuning work involved wrestling with locks introduced into the system; not surprisingly, one of the more challenging locks was our equivalent of the address-space lock for the VMM. This performance work continues to this day as physical systems are deployed with ever greater numbers of cores and as these systems cope with scale through the use of NUMA architectures.

Mere performance problems were only one of the kinds of challenges we met. We also had to learn how to address races in guests exacerbated by the timing characteristics of VMs. Examples were quickly found in a variety of early Windows releases, Netware, Solaris, and some Linux distributions. Our techniques included controlling the guest's perception of time, and more bizarre techniques such as "lock step" execution in which VCPUs go round-robin, executing one guest instruction at a time, to make it safely through race-prone regions of guest code: after you, my friend!

How one structures any system to support concurrency has a profound impact not just on performance and scalability but also on how tractable the system is to reason about and work with, and how it evolves over long time periods where requirements may change. In the remainder of this section, we discuss synchronization mechanisms and the vSMP MMU.

4.1 The vSMP Programming Model

Since the physical CPUs in an SMP system execute largely independently of each other, it is natural to model VCPUs as independent threads of execution. These threads must coordinate when accessing shared device state, when updating gPA-to-hPA mappings, and when handling trace events. The coordination must present a consistent view to the guest while being fast and scalable.

Initially, we used Posix-like synchronization mechanisms, including mutexes, semaphores, and condition variables. However, we soon found that in order to control when and where VCPUs handled certain events, we needed additional higher-level mechanisms: crosscalls, actions, and stop.

In cases where this coordination requires synchronicity, the VMM employs *crosscalls*. Crosscalls allow one VCPU to request some service or action from other VCPUs. The initiating VCPU blocks until the request has been handled by all the callees.

To understand the motivation for crosscalls, first consider traces in a vSMP VMM. The set of pages that are traced is dynamic. Adding a page to this set (installing a trace) requires all VCPUs to respect the presence of the trace before the installer can depend upon it. Another use of crosscalls is the invalidation of gPA-to-hPA mappings in order to allow memory to be shared, unshared, swapped or remapped [19].

To reason about crosscalls and their interaction with mutexes, we statically *rank* crosscalls and mutexes based on the kinds of resources they manipulate or guard. VCPUs can progress from one ranked operation to another only in order of increasing rank. Additionally, any VCPU blocking on a mutex or invoking a crosscall must serve same- or higher-ranked incoming crosscalls. These simple constraints, checkable by assertions, ensure that deadlocks do not occur.

In cases where coordination between VCPUs only requires some eventual processing on the part of other VCPUs (i.e., constitute an asynchronous request), the VMM employs *actions*. Actions, once delivered to a VCPU, are guaranteed to be processed before the execution of the next guest instruction, but the caller cannot make any assumptions about how long delivery will take. For example, we use actions to remove traces. While we could have used a synchronous crosscall, an asynchronous action permits looser coupling and therefore better scalability.

Our final higher-level coordination mechanism is *stop*: bring all VCPUs to a stand-still at base rank and at a virtual instruction boundary. Having "stopped the world," the VMM can perform operations that are safe only in this controlled environment. For example, we use a form of stop to checkpoint a virtual machine's state.

Over the years, we found these higher-level coordination mechanisms, and especially ranks, to serve our needs well. However, they were no silver bullet: we still had to program carefully to attain scalability, including sometimes using lock-free algorithms.

4.2 vSMP MMU Virtualization

In x86 SMP systems, each core has its own page walker and TLB. However, most operating systems use a shared pool of memory for page tables. To bridge from shared page tables to per-core TLBs, a precisely coordinated "TLB shootdown" approach must be used to invalidate mappings.

Given the dichotomy between shared page tables and per-core TLBs, how should the virtual MMU be designed? On the one hand, shadow page tables can be seen as counterparts to primary page tables, suggesting that we share shadow page tables between VCPUs. On the other hand, shadow page tables have similarities with TLBs in that they cache mappings, suggesting use of per-VCPU shadow page table caches.

For simplicity and to generally decouple VCPUs, we went with per-VCPU shadow page table caches. Reasoning about nonshared caches seemed easier, although we knew we would have to watch out for potential drawbacks including: (1) increased memory usage when multiple VCPUs would all shadow the same primary page table, and (2) more hidden page faults than if VCPUs cooperate to validate the working set into shared shadows, especially when guest schedulers migrate processes from VCPU to VCPU.

Our initial vSMP MMU was noncaching. We simply ran independent instances of the familiar shadowing algorithm on each VCPU. Each VCPU would, independently, take hidden faults, build up private shadow page tables, and fully clear these shadow page tables on context switches. The noncaching MMU allowed us to run and examine the first set of SMP workloads.

For higher performance, we needed caching to allow shadow page tables to survive context switches. And, as in the uniprocessor case, we would need traces to keep the shadow page tables in sync with the primary page tables. Our first caching vSMP MMU used the trace system directly. Each time a VCPU processed a hidden fault, it installed a write trace on the path that the page walker traversed in the primary page tables to satisfy the hidden fault. The trace subsystem tracked which VCPUs had a trace on which pages, allowing for precise notification to the VCPUs that were affected by page table writes. As expected, the caching MMU lowered the number of hidden faults and gave us a significant performance boost.

Trace firing was the next optimization target. To fire an MMU trace, we crosscalled all affected VCPUs. This synchronicity was both expensive and unnecessary: TLB consistency rules only require that page table updates are reflected at the next TLB flush. Thus, instead of carrying MMU trace events with synchronous crosscalls, we implemented an inbound "patch set" queue for each VCPU. To notify another VCPU of a trace event, the event could simply be added to its patch set. Each VCPU would process the queued events in its patch set at the next TLB flush (or sooner if the fixed-size data structure filled up). This decoupling of trace producer and consumer improved our MMU performance substantially.

Finally, to reduce the number of hidden faults resulting from remotely produced traces, we added eager validation into the patch set processing.

The combined effect of these optimization steps gave us a software MMU with acceptable performance, albeit one that we would continue to refine and tune over the years.

5. VIRTUALIZING 64 BIT x86

AMD started shipping x86 compatible 64 bit Opteron CPUs in 2003. Their architecture—called AMD64, x86-64, and later simply x64—added a new 64 bit execution mode, *long mode*, while retaining existing 16 and 32 bit execution modes for complete x86 backwards compatibility.

During those years, server workloads were increasingly pushing against memory limitations. PAE mode extended the physical address space from 4 GB to 64 GB, but did nothing to relieve virtual address space pressure in the increasingly "cramped" 32 bit environment: any individual process could still address at most 4 GB of virtual memory at a time. Long mode, however, provided 64 bit addressing (though implementations could cap at 48 usable bits), thereby eliminating address space pressure completely. In long mode, 4-level hierarchical page tables map 48 bits of virtual addresses to, typically, 48 bits of physical addresses.

Long mode also brought instruction set improvements, extending the eight legacy x86 registers to 64 bits in much the same way as the 80386 processor had widened the registers from 16 to 32 bits almost two decades earlier. For example, the 16 bit `%ax` register had grown into the 32 bit `%eax` in 1985, and in 2003 it grew to become the 64 bit `%rax` register. Moreover, long mode added an additional eight general-purpose registers to alleviate one of the most often cited architectural bottlenecks of the x86 architecture.

AMD also took the opportunity to remove or simplify some legacy features in long mode. Unfortunately, two of these "improvements" directly affected our ability to use BT to virtualize x64:

- All segment registers but `%fs` and `%gs` were flattened (base zero, maximal limit). For `%fs` and `%gs`, only the base address functionality remained, i.e., limit checks were removed. The motivation was, probably, that the base address functionality on two segment registers sufficed for providing per-thread storage with segment registers. Unfortunately, without limit checks, a BT-based VMM could no longer use segmentation to protect itself from the guest.
- The `lahf` and `sahf` instructions were removed from long mode. These instructions move ALU-flags to and from the 8 bit `%ah`, respectively. For a binary translator, they provide a faster way to save and restore flags than the alternative instructions, `pushf` and `popf`, that move flags to and from the stack.

Lacking `lahf`/`sahf` would be an inconvenience, but lacking segment limit checks made building a high performance BT-based 64 bit VMM much harder. Fortunately, AMD saw enough potential value in having a 64 bit virtualization solution that they added support for segment limit checks and `lahf`/`sahf` in "Revision D" of the processor.

5.1 The Peer Model

With the necessary architectural support secured from AMD, we set out to virtualize long mode in much the same manner as we had previously virtualized the 16/32 bit modes. But, along the way, we had to make a few intermediate stops.

We got an Opteron prototype system (in a large box with rollerskate wheels). First, we installed a 32 bit host OS on it, and verified that our existing 32 bit VMM could run VMs on it. As expected, this step was easy: the x64 architecture was indeed backwards compatible with 32 bit x86. Let us denote this configuration $\frac{32}{32}$.

Next, we installed a 64 bit host OS, and began pondering how to write the "world switch code" that would get from a 64 bit host to a 32 bit VM running on our 32 bit VMM, i.e., $\frac{32}{64}$. We approached this challenge with some trepidation: back in the 80286 days, CPUs could switch from old-style 8086 real mode to new-style 80286 protected mode, but they had no reliable way to get back! Might there be a similar problem lurking in the transition between 32 and 64 bit protected mode? Fortunately, while this mode switch was delicate, involving taking the CPU all the way back to 16 bit real mode and running instructions on an identity-mapped page (i.e., one with the same virtual and physical address), this all worked out. Soon, we had a 32 bit VM running on our VMM with the host in 64 bit mode. The CPU happily transitioned between long mode and legacy mode thousands of times per second.

Feeling confident from this experience, we set out in earnest to virtualize long mode, i.e., implement the $\frac{64}{32}$ and $\frac{64}{64}$ configurations. Since we already had a 16- and 32-bit capable VMM, we decided to write a new long-mode capable VMM that could do just that: virtualize x64 long mode. We would use this "vmm64" when the guest was running in long mode and use our existing VMM, now renamed "vmm32" when the guest was running 16 or 32 bit code. This decision allowed us to reuse our existing VMM largely unchanged while keeping long mode functionality in a new binary. We hoped this would improve performance and keep the task ahead of us simpler. Soon we started using the term "peers" to describe the (vmm32, vmm64) pair that would cooperate to run a 64 bit guest.

In addition to promising code reuse, the peer model assured us that 32 bit code would execute in exactly the same manner, whether the guest was potentially using long mode on an x64 CPU or using legacy mode on a 32 bit legacy CPU. In a commercial setting where paying customers commit mission critical workloads to our platform and require extremely high reliability, avoiding duplicated functionality is not just attractive, but almost a requirement.

While we would ship two separate binaries, one 32 bit and one 64 bit, we still wanted as much *source code* shared between the two as possible. We started thinking of our source code as having three flavors: 32 bit, 64 bit, and common code. The common code would be compiled twice: once into the vmm32 binary and once into the vmm64 binary. Initially, 100% of our source code was 32 bit, and 0% was common and 64 bit. By the time we shipped our first 64 bit capable product, VMware Workstation 5.5 in 2005, the breakdown was approximately 15% 32 bit specific source, 15% 64 bit specific source, and 70% common source. We consider this breakdown a successful outcome: while we had two binaries, each just a few hundred KBs, there was very little source code duplication. (It is not, however, the end-stage; more on that later.)

As with the existing vmm32, the new vmm64 would use BT for privileged code, and direct execution for user code. When running translated code, the VMM would use the "reintroduced" (long mode) segment limit checks to protect the VMM's address space.

The first step was to write a 64 bit peer that could merely reflect interrupts back to the host, while not even trying to run the guest. This required us to use the world switch code from our earlier $\frac{32}{64}$ milestone if the host OS was 32 bit, or a 64-to-64 bit version for a 64 bit host OS.

To bring up the 64 bit peer, we repurposed our $\frac{32}{64}$ switch technology one more time, now wrapping it into a minimal toy OS that would start in real mode, reach 32 bit protected mode, then switch to long mode and run a handful of instructions before going back to 32 bit mode and shutting down. Virtualizing this minimal OS kept us busy for weeks, forcing us to deal with long mode paging, descriptor tables, control registers, and, of course, 64 bit BT in general.

Drawing upon experience from our 32 bit VMM, we knew we wanted to have a full 64 bit interpreter as our guaranteed-to-be-complete execution mechanism. It would deal with the most contrived corner cases like a page fault on an access straddling a page boundary in the middle of a task switch. In other words: CISC fun. The interpreter would be our complete mechanism, giving us the confidence to write a speed-focused binary translator. We would have one very, very complete path and one very fast path.

Dividing ourselves into two subteams, one team set out to write the translator, the other to write the interpreter. The next day (!), the translator team reported back: we have a *complete* translator. Indeed, it was complete; every translation called into the interpreter with a decoded instruction: "please run this one." Now pressure was on the interpreter team. Soon, they got to the point where the minimal test could run. Meanwhile, the translator team gradually shifted the burden away from the interpreter by adding "real" translations for IDENT instructions, branches, calls, returns, etc. (in rough order of dynamic frequency).

Having run the long mode equivalent of "hello world," the next challenge was clear: 64 bit Linux and Windows. The details of getting from the minimal test OS to a real OS were overall unsurprising. One particularly memorable moment was the day we got our first 64 bit blue-screen of death (BSOD) full of amazing 64 bit hex numbers. We happily printed it out, posted the picture on our door, and sent mail with a screen shot to "all," declaring victory.

Indeed, from this point on, victory was in sight. We got Linux and XP to boot. We added optimizations to enable descriptor table shadowing, a caching MMU, direct execution, and soon we found that 64 bit guests ran as well as, if not better, than 32 bit guests ever had. The "better" part we ascribe to a cleaner 64 bit architecture and less address space pressure (so less use of memmap and other operations that suffer from overheads in a virtual environment).

5.2 Peer Elimination

While peers allowed us to reuse existing code, and focus on the new long mode separately from legacy modes, peers contributed their own complexity. Whenever the guest would switch execution mode, for example in response to taking a System Management Interrupt out of long mode into 16 bit real mode, the VMM would need to switch from the vmm64 peer to the vmm32 peer. The peer switch itself was delicate and somewhat slow, but fortunately also infrequent. Complexity also resulted from the management of peer-shared state, including virtual device and VCPU state, as well as VMM implementation state such as trace data structures and shadow page tables. Thus, as 64 bit CPUs became more prevalent, it became appealing to run the VMM exclusively in long mode, i.e., have just one peer, vmm64.

To always run in long mode, we extended the 64 bit translator to accept 16 and 32 bit guest code input but produce 64 bit output, a form of translation we call "widening" [8]. Returning to our 32 bit spin lock example from Section 3.1, widening translation produces this 64 bit output:

```
* lea     -4(%rsp),%esp        ; PROT_PUSH
  mov     %ebx,(%rsp)
  mov     %eax,%edx            ; IDENT
  and     $0xfd,vcpu.flags     ; PRIV
  mov     $1,%ecx              ; IDENT
  xor     %ebx,%ebx
* mov     %ebx,%eax            ; IDENT
  lock
  cmpxchg %eax,%ecx,(%edx)
  test    %eax,%eax
  jnz     [spin]               ; JCC
* mov     (%rsp),%rbx          ; WIDEN_MEM
  lea     4(%rsp),%esp
* mov     (%rsp),%r11d         ; RET_LAUNCH
  mov     %rcx,%r8
  movzx   %r11b,%ecx
  lea     4(%rsp),%esp
  jmp     rtc(8*%rcx)
```

The first TU begins with the translation of a push instruction. The 32 bit translation used an IDENT translation for push, but 64 bit mode on x86 only allows for 64 bit stack operations. Since the guest expects only 32 bits to be written to the stack, the widening translator must break the push into an update of the stack pointer followed by a write to memory. The stack pointer adjustment uses a "load effective address" `lea` instruction, which can perform addition, but, unlike an `add` instruction leaves ALU-flags undisturbed. In this case, we use `lea` to subtract four from `%rsp`, storing the result into `%esp` (i.e., zeroing the high 32 bits of `%rsp`).

The memory accessing `cmpxchg` is translated IDENT just like in 32 bit mode, but operates a bit differently. Unlike a 32 bit guest on a 32 bit VMM, and a 64 bit guest on a 64 VMM, with a 32 bit guest on a 64 bit VMM, we have plenty of free address space. Accordingly, we place the 64 bit VMM above 4 GB, i.e., above the highest address that a 32 bit guest can generate. This ensures (1) that there is no overlap between guest and VMM address space usage (and therefore no performance cliff), and (2) that there is no need for segment limit checks to enforce the guest/VMM boundary.

To confine translated code to the low 4 GB of the address range, the translator can apply a prefix on memory references to clip addresses from 64 bit to 32 bits. By leaving out the prefix the translator still has an escape mechanism to reach VMM data structures just as the `%gs` prefix allowed when using segment limits. The `cli` translation above no longer needs the `%gs` prefix that it had in the 32 bit translation. Moreover, by setting up a guard page at 4 GB, the translator can often get away with eliminating the address prefix. For example, if the translator can prove that the high 32 bits of `%rax` are zero, a memory reference like `(%eax)` can be translated to `(%rax)`, which is essentially as good as IDENT.

The above example came from a guest using flat segments. Guests that use non-flat segments, including when running in real mode, require the widening translator to do limit checks and base offsetting in "software." This results in longer and slower translations, but such non-flat code is rare and typically designed to run on slower CPUs (e.g., OS/2 designed to run on a 33 MHz 80386), so the overheads are affordable on fast modern CPUs.

We first shipped vmm64 in 2005, but only now, almost five years later, has the time come when we can move all functionality into vmm64 and eliminate vmm32. We had to wait for vmm64 to gain the required functionality, but more importantly, we had to wait for 64 bit CPUs to become ubiquitous so that we have within sight the day we can retire support for vmm32. Otherwise, we would find ourselves in the undesirable position of supporting legacy guest execution with two different implementations, one in each peer.

5.3 Intel's EM64T

The 64 bit x64 architecture originated with AMD's Opteron. In 2004, just a year later, Intel shipped their 64 bit extension of x86, calling it IA-32e or EM64T. This architecture is surprisingly similar to AMD's, including the removal of `lahf`/`sahf` and segment limit checks. While Intel later added back `lahf`/`sahf`, and thereby enabled widening BT, Intel never reintroduced segment limit checks in long mode. To this date, to run a 64 bit guest on an Intel CPU using VMware's software, the CPU must have VT-x hardware support for virtualization.

6. HARDWARE SUPPORT

By 2005, both Intel and AMD were publicly discussing architectural extensions to enable trap-and-emulate virtualization of the x86 architecture. These extensions, named VT-x and AMD-V, respectively, differ in details but their overall designs are similar and they both enable x86 instruction set virtualization without the need for BT [3, 10]. We review this "first generation" hardware support in Section 6.1.

Subsequently, in 2007-8, additional hardware support for virtualization, now seeking to eliminate the need for shadow page tables, was brought to market: AMD added Rapid Virtualization Indexing (RVI, originally named Nested Page Tables, NPT) and Intel added Extended Page Tables (EPT). Again, while differing in names and details, the two vendors' architectures are remarkably similar overall. We review this "second generation" hardware support in Section 6.2.

6.1 Instruction Set Virtualization: VT-x and AMD-V

Since VT-x and AMD-V differ only in details, we describe the hardware support using common terminology. The anchor point is an in-memory data structure, which we refer to as a *virtual machine control block* (VMCB). The VMCB combines control state with a subset of the guest VCPU state. A new, less privileged execution mode, *guest mode*, supports direct execution of guest code, including privileged kernel code. To contrast with guest mode, we refer to the previously architected execution mode as *host mode*.

A new instruction, `vmrun`, transfers from host to guest mode. Upon execution of `vmrun`, the hardware loads guest state from the VMCB and continues execution in guest mode. Guest execution proceeds until some condition, expressed by the VMM using the control bits of the VMCB, is reached. At this point, the hardware performs an *exit* operation, which is the inverse of `vmrun`: guest state is saved to the VMCB, VMM state is loaded, and execution resumes in host mode, now in the VMM.

To aid the VMM in handling the exit, the hardware writes certain *exit code* fields in the VMCB. For example, exits due to guest `in`/`out` instructions provide the port number, width, and direction of the operation whereas exits due to page faults provide the faulting address and access mode. After emulating the effect of the exiting operation in the VMCB, the VMM again executes `vmrun` to resume execution of the guest.

Within guest mode, the VM can issue system calls to switch between kernel and user mode, use segmentation, run instructions like `popf` that are sensitive to privilege level, change code size between 16, 32 and 64 bit code, and take faults without causing any exits to the VMM.

For some instructions, such as `cpuid`, an exit is mandatory. For many others, the VMM can choose whether to interpose by setting control bits in the VMCB. For example, at times, VMM may request an exit when the guest attempts to execute the `rdtsc` ("read time-stamp counter") instruction to help the VMM virtualize time and timers with greater accuracy; at other times, the higher performance permitted by exit-free execution of `rdtsc` may be desirable.

The first generation hardware support provides a fairly complete virtualization solution, leaving the VMM with the task of vectoring to emulation code based on exit codes provided in the VMCB. Most of this emulation code remains the same, whether instruction execution is done with BT or hardware support. The emulation code includes peripheral device models, code for delivery of guest interrupts, and infrastructure tasks such as logging, synchronization and interaction with the host platform. Since first generation hardware support lacks explicit support for memory virtualization, the VMM must implement a software MMU using shadow page tables. However, unlike the BT case, the address space defined by the shadow page tables is active only when running in guest mode. This gives the guest access to its entire address space, while the VMM can have a full address space of its own. The price for this "convenience" is a context switch on each `vmrun` and exit.

The performance characteristics of the hardware-assisted VMM differ substantially from the BT-based VMM. With hardware-assist, the guest runs at full speed, unless an exit is triggered. Thus, virtualization overheads are determined as the product of the exit frequency and the average cost of handling an exit.

Early implementations of hardware support had quite high exit costs. For example, the Intel P4 model 672 CPU took 4340 cycles for a guest to VMM to guest "null" round-trip [1]. Later processors have reduced these costs by almost an order of magnitude, but exits, at several hundred cycles each, are still far from free. In addition, one must add in software costs for actually handling the exit. This leaves us with a situation where reducing the frequency of exits is the most important optimization.

To help avoid the most frequent exits, VT-x and AMD-V include ideas similar to the IBM System 370 *interpretive execution* facility [11]. Essentially, certain types of exits can be "buffered" in the VMCB rather than unconditionally causing exits. Consider `popf`. A naive extension of x86 to support trap-and-emulate virtualization would trigger exits on all guest mode executions of `popf` to allow the VMM to update the virtual "interrupts enabled" bit. However, guests may execute `popf` very frequently, leading to an unacceptable exit rate. Instead, the VMCB includes a hardware-maintained shadow of the guest `%eflags` register. When running in guest mode, instructions operating on `%eflags` operate on the shadow, removing the need for exits.

The exit rate is a function of guest behavior, hardware design, and VMM software design: a guest that only computes never needs to exit. Hardware provides means for throttling some exit types, and VMM design choices, particularly the use of traces and hidden page faults, directly impact the exit rate.

6.2 Memory Virtualization: RVI and EPT

To drive a software MMU, a VMM using first generation hardware support must interpose on several guest events:

- the VMM must write-protect primary page tables to trigger exits when the guest updates primary page tables so that the VMM can propagate the change into the shadow page tables (e.g., invalidate).
- the VMM must request exits on page faults to distinguish between hidden faults, which the VMM consumes to populate shadow page tables, and true faults, which the guest consumes to populate primary page tables.

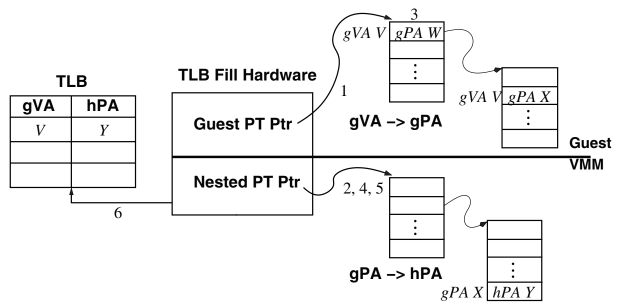

Figure 3: Nested paging hardware.

- the VMM must request exits on guest context switches so that it can activate the shadow page tables corresponding to the new context.

We learned the importance of fast trace handling from years of work with the BT-based VMM. Unfortunately, the main tool to speed up traces, adaptive BT, does not carry over to hardware-assisted virtualization. With the resulting higher trace costs, and little flexibility to "work" the three-way tradeoff outlined above, it came as no surprise that first generation hardware support often did not outperform a BT-based VMM [1].

We speculate that this situation, as well as experience from IBM's s/370 machines [11] motivated AMD and Intel to implement a second generation of hardware support, aimed directly at memory virtualization.

Both AMD's RVI and Intel's EPT hardware support for memory virtualization require use of their respective forms of hardware support for instruction virtualization. In other words, one can use AMD-V/RVI and VT-x/EPT, but neither RVI nor EPT can be combined with BT.

To use RVI or EPT, the VMM maintains a hardware-walked "nested page table" that translates gPAs to hPAs. This mapping allows the hardware to dynamically handle guest MMU operations, eliminating the need for VMM interposition. The operation of this scheme is illustrated in Figure 3.

While running in guest mode, the TLB contains entries mapping gVAs all the way to hPAs. The process of filling the TLB in case of a miss is more complicated. Consider the case of a guest reference to virtual address V that misses in the hardware TLB:

1. The hardware uses the guest page table pointer `%cr3` to locate the top level of the guest's hierarchical page table.
2. The guest's `%cr3` contains a gPA, which must be translated to a hPA before dereferencing. The hardware walks the nested page table for the guest's `%cr3` value to obtain the hPA of the top level of the guest's page table hierarchy.
3. The hardware reads the guest page directory entry corresponding to gVA V yielding a gPA W.
4. The gPA read from the page table entry in step 3 must now be translated via the nested page table before proceeding. Steps 3 and 4 repeat once for each level in the guest page table (up to four levels for a long mode guest).
5. Having discovered the hPA of the final level of the guest page table hierarchy, the hardware reads the guest page table entry corresponding to V. In our example, this page table entry points to gPA X, which is translated via a final walk of the nested page table, e.g. to hPA Y.
6. The translation is complete: gVA V maps to hPA Y. The page walker can now fill the TLB with an appropriate entry (V, Y) and resume guest execution, all without VMM intervention.

With this hardware support, *all* the exit-related costs associated with the software MMU go away: there are no trace-induced exits, no context-switch exits, and no hidden/true fault exits. Moreover, the VMM does not have to allocate memory for shadow page tables, reducing memory usage.

However, these benefits do not come for free. The cost to service a TLB miss *will* be higher with nested paging than without. In a naive implementation, without caching, the number of steps required to fill the TLB is quadratic in the depth of the page tables: the nested page table must be walked for each level of the guest page table. Fortunately, as described by Bhargava et al. [6], locality properties make it possible to effectively cache data required by most of these steps, reducing the TLB miss costs substantially. Moreover, use of large pages (2 MB and 1 GB) in the guest and VMM page tables can both reduce the number of TLB misses and speed up the handling of the ones that remain.

For workloads that execute mostly in a single static address space, such as a thread-based Java workload, there are few performance benefits to nested page tables (other than the reduction in memory usage) since such workloads have little trace activity, few hidden faults, and few context switches. However, for many other workloads, including ones that fork short-lived processes (e.g., some web servers and `make` jobs), avoiding software-MMU related exits more than outweighs the cost in TLB misses with nested paging. All in all, our experience with second generation hardware support has been favorable; it usually yields better performance, and – as importantly – it significantly reduces performance cliffs.

7. NESTED VIRTUALIZATION

The idea of nested virtualization, i.e., running a virtual machine in a virtual machine, goes back to the mainframe era and has now re-emerged in the x86 world [5, 12].

We found the first use case within our own engineering organization: just as an OS developer benefits from the controlled development environment, offered by VMs, so can a hypervisor developer benefit from a controlled development environment *that permits nesting of virtual machines*. For example, we have been able to use nested virtualization to debug "host crashes."

We found the next use case in our QA organization: by testing our vSphere management software against virtualized ESX hosts, we can scale the test environment at lower hardware costs by having each physical host run several ESX VMs.

We also found use cases in sales, where one can demo a full vSphere setup with multiple virtualized hosts and management software by running VMs on a laptop, and training, where one can practice installation, setup, and work flow with virtual ESX hosts.

However, the most compelling reason why nested virtualization is inevitable, even in production environments, comes from observing that recent versions of popular operating systems now use virtualization internally, either to establish compatible environments (e.g., XP mode on Windows 7) or to partition resources. Running such guests requires nested virtualization to use all their features.

Given all these use-cases, what does it take to run a nested virtual machine with reasonable performance? First, note that our VMM offers several different execution modes. Some nest more readily than others. For example, it is relatively straightforward to run a BT-based VMM inside a VM that runs on a VMM using VT-x/EPT or AMD-V/RVI. The BT-based VMM needs no special hardware features to support virtualization: it is just an x86 program like any other. In contrast, running a BT-based VMM on top of a BT-based VMM is prone to suffer inefficiencies. For example, the outer VMM will see the inner VMM's TC as a "program" that uses self-modifying code extensively. Self-modifying code suffers performance overheads as the outer VMM needs to invalidate translated code whenever its "source" code changes. Worse still, the inner VMM will (unless modified for the purpose of nesting) tend to use addresses that overlap the outer VMM, causing much more frequent guest/VMM collisions than when running "normal" guests.

More interestingly, both VT-x and AMD-V can "self-virtualize." That is, when a VMM runs on a physical CPU with this hardware support for virtualization, it is possible, with some effort, to present a virtual CPU to the guest that has the same features. In practice, performance is much better when going one step further and virtualizing VT-x/EPT or AMD-V/RVI: the lower exit frequency of the inner VM that runs with EPT or RVI will result in much less overhead from nested virtualization.

Virtualizing VT-x and AMD-V highlighted a performance difference between the two architectures. With AMD-V, the VMCB has a defined in-memory layout that the VMM accesses using plain loads, stores, etc. Because VMCB accesses use normal memory operations they run at full speed even for a nested VMM. With VT-x, the VMCB is an opaque data structure that must be accessed using special purpose `vmread` and `vmwrite` instructions. This layer of abstraction gives the processor freedom to change the layout of the VMCB or implement special purpose caching. But with nested VT-x, each time the inner VMM uses `vmread` or `vmwrite` it generates an exit to the outer VMM. A nested `vmread` or `vmwrite` therefore runs about two orders of magnitude slower than natively. To get good performance with nested VT-x, the (inner) VMM must be written to use as few of these instructions as possible per exit.

Without the ability to run nested VMs, virtualization could have fallen victim to its own success. While nested virtualization is still in the early stages, we are hopeful that the technology will work well. And, as work progresses towards this goal, nested virtualization has already proven itself to be a source of brain teasers and fun problems to debug!

8. OTHER TOPICS

To stay within a reasonable length, this paper had to omit discussion of several interesting features of the VMM. We briefly list the most important functionality here.

A VMM is responsible for virtualizing timers in such a manner that the virtual machine sees a "sufficiently realistic" view of the world that it will run correctly. Timers include devices that can generate periodic or one-shot interrupts as well as devices that measure elapsed time. In particular, the VMM must keep these different time sources consistent with each other, across multiple VCPUs, including under circumstances of overcommit where a particular VM may be descheduled for periods of time. Solving this problem well, such that guests can track time-of-day with an accuracy of a few parts per million, and measure elapsed time accurately down to fractions of a second, is a surprisingly hard problem [18].

Our VMM implements a record/replay capability that allows the entire virtual instruction stream of a VM to be captured in a form that can be replayed deterministically, i.e., with instruction-for-instruction accuracy [21]. While restricted to uniprocessor VMs, this capability forms the basis for a replay-based debugger that makes non-deterministic bugs deterministic (once recorded) and provides a reverse execution capability, greatly helping the debugging of software running in VMs.

The same record/replay technology supports a fault tolerance solution: by recording the execution trace of a "primary" VM and transmitting it over a network to a different host where a "backup" VM replays it, the backup VM can act as a hot standby, ready to take over should the primary crash. Since the replaying VM's execution matches the recording one's with instruction-level precision, fail-over can be done at any time without any data loss [16]. The key value of this approach to fault tolerance comes from the fact that it is much more network-bandwidth efficient to transmit the recording of VM execution from the primary to the backup and regenerate the VM state than directly transmit the state that has changed.

The VMM implements support for managing the memory working sets of guests including techniques to support overcommitment and remapping of memory, such as ballooning, swapping, transparent page sharing [19] as well as NUMA-migration when VMs are rescheduled. This cooperation is necessary because while the vmkernel specifies what must be achieved, the VMM knows how and where the memory is in use.

9. CONCLUSIONS

Over the past twelve years, we have witnessed x86 virtualization proceed from a technical curiosity to the foundation for a new way of organizing data centers and information technology. To facilitate this, the VMM evolved from a 32 bit software system focusing on desktop workloads to a 64 bit, vSMP-capable platform that can run even the most challenging server workloads.

Twelve years is a long time, but is software life measured in dog years or human years? Does the VMM today feel old or is it ready to take on new challenges? Overall, we see things positively. While large changes were implemented over the years and today's VMM shares little code with the one that started it all, the overall design remains minimal and appropriate for the task at hand. By constantly retiring obsolete code and improving the architecture we continue to have a VMM with little legacy code or accretion of cruft: the VMM is still only a few hundred kilobytes in size. Our guiding philosophy is that each simplification, such as the recent elimination of peers, gives us a renewed "complexity budget" that we can spend to meet new challenges.

Two contemporary challenges stand out. First, VMs must continue to grow with the workloads being deployed. This means keeping up with the virtual version of Moore's law by adding VCPUs and memory at the rate the corresponding physical resources, i.e., cores and RAM, grow. Second, we must further expand the types of workloads applicable to virtualization, including latency- and time-sensitive workloads, developer workloads (e.g., profilers and debuggers), and workloads written in new language frameworks, perhaps spanning multiple VMs.

It is unusual for any technology to have as large an impact as x86 virtualization has had. Which circumstances came together to make it happen? Crucially, VMware started with a really good idea, x86 virtualization, at the right time, when x86 servers had become so powerful that most deployed servers were underutilized. Additionally, VMware had a bold and convincing vision so it could attract an exceptionally strong team of software engineers who set out to perfect the VMM as well as other software modules. For several years, VMM development focused on finding effective, simple and elegant solutions, using the x86 architecture to its fullest extent and often in ways unanticipated by the x86 architects themselves. The combination of talent, perseverance, generality of the x86 architecture, a clear value-proposition in improved efficiency and flexibility, and a healthy dose of luck, succeeded in creating a significant virtualization market opportunity. As a result, a new force came into play when x86 CPUs acquired architectural extensions to directly support virtualization. In turn, the market opportunity continued to expand, while VMM development shifted from being purely software-based to employing a mix of software and hardware techniques to optimize virtual machine execution. We expect this co-evolution of hardware and software to continue as virtualization is extended to a wider and wider set of uses.

While this paper looks back at what we have learned and done, we are looking forward.

10. ACKNOWLEDGMENTS

First, and most importantly, we express our sincerest thanks to the five VMware founders: Diane Greene, Mendel Rosenblum, Edouard Bugnion, Edward Wang, and Scott Devine. Their vision and effort changed our careers and industry, and now, 12 years later, made this paper possible.

The VMM benefitted from the work and inspired insights of many talented individuals over the years. Throughout this paper, the authors have used "we" to refer broadly to the whole monitor group and not narrowly to the authors. In approximate arrival order, "we" are:

Mendel Rosenblum, Edouard Bugnion, Scott Devine, Jeremy Sugerman, Bich Le, Beng-Hong Lim, Luigi Semenzato, Ole Agesen, Jennifer Anderson, Pratap Subrahmanyam, Jeffrey Sheldon, Keith Adams, Ganesh Venkitachalam, Wendy Chien, Mike Chen, Robert Manchester, Matthias Hausner, Taylor Hutt, Michel Lespinasse, Boris Weissman, Steve Herrod, Daniel Arai, Vyacheslav Malyugin, Sahil Rihan, Alberto Munoz, Alan Shieh, Mike Cohen, Vladimir Kiriansky, Doug Fawley, Michael Devine, Sriram Sankar, Jack Lo, Steve Muckle, Garrett Smith, Geoff Pike, Sergey Lyubskiy, Zach Amsden, Ariel Tamches, Ravi Prasad, Jim Mattson, Eli Collins, Mike Cui, Alex Garthwaite, Dan Hecht, Alex Moshchuk, Meenali Rungta, Bhavesh Mehta, Ka Wing Ho, Carlos Robles, Rohit Jain, Mike Rieker, Robert Benson, Yury Baskako, Charles Hannum, Min Xu, Alex Mirgorodskiy, Jeff Glasson, Wei Xu, Doug Covelli, E Christopher Lewis, Gautam Altekar, Joyce Chen, Alexander Klaiber, Ross Knippel, Michael Paleczny, Peter Desnoyers, Radu Rugina, Kapil Arya, Wing-Chi Poon, Alok Kataria, Ron Mann, Irene Zhang, Karen Zee, Jerri-Ann Meyer, Benjamin Serebrin, Palash Agarwal, Ricardo Gonzales, Patrick Chiu, Joyce Spencer, and many others from all of VMware.

We also thank Regis Duchesne, Tim Mann and Alex Protopopescu for comments on drafts of this paper.

11. REFERENCES

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, 2006.

[2] O. Agesen. Binary translation of returns. In *Workshop on Binary Instrumentation and Applications*, pages 7–14, October 2006.

[3] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, June 2010. Chapter 15.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on programming language design and implementation*, pages 1–12, 2000.

[5] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: Design and implementation of nested virtualization. In *OSDI '10: 9th USENIX Symposium on Opearting Systems Design and Implementation.* USENIX Association, 2010.

[6] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, 2008.

[7] P. E. Ceruzzi. History of digital computers. In A. Ralston, E. D. Reilly, and D. Hemmendinger, editors, *Encyclopedia of Computer Science*, pages 545–570. Nature Publishing Group, Londo, UK, 4th edition, 2000.

[8] Y. Chen. Dynamic binary translation from x86-32 code to x86-64 code for virtualization. Master's thesis, Massachusetts Institute of Technology, 2009. http://hdl.handle.net/1721.1/53095.

[9] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on measurement and modeling of computer systems*, pages 128–137, 1994.

[10] Intel Corporation. *Intel® Virtualization Technology Specification for the IA-32 Intel® Architecture*, April 2005.

[11] D. L. Osisek, K. M. Jackson, and P. H. Gum. ESA/390 interpretive-execution architecture, foundation for VM/ESA. *IBM Systems Journal*, 30(1):34–51, 1991.

[12] W.-C. Poon and A. K. Mok. Bounding the running time of interrupt and exception forwarding in recursive virtualization for the x86 architecture. Technical Report VMware-TR-2010-003, VMware, Inc., 3401 Hillview Avenue, Palo Alto, CA 94303, USA, Oct 2010.

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

[14] G. J. Popek and C. S. Kline. The PDP-11 virtual machine architecture: A case study. In *Proceedings of the fifth ACM symposium on Operating systems principles*, SOSP '75, pages 97–105, 1975.

[15] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2000. USENIX Association.

[16] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *SIGOPS Oper. Syst. Rev.*, 44(4), 2010.

[17] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2001.

[18] VMware. *Timekeeping in VMware Virtual Machines*, May 2010. http://www.vmware.com/vmtn/resources/238.

[19] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[20] L. D. Wittie. Microprocessors and microcomputers. In A. Ralston, E. D. Reilly, and D. Hemmendinger, editors, *Encyclopedia of Computer Science*, pages 1161–1169. Nature Publishing Group, London, UK, 4th edition, 2000.

[21] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.