A Comparison of Software and Hardware Techniques for x86 Virtualization

Keith Adams
VMware
kma@vmware.com

Ole Agesen
VMware
agesen@vmware.com

Until recently, the x86 architecture has not permitted classical trap-and-emulate virtualization. Virtual Machine Monitors for x86, such as VMware® Workstation and Virtual PC, have instead used binary translation of the guest kernel code. However, both Intel and AMD have now introduced architectural extensions to support classical virtualization.

We compare an existing software VMM with a new VMM designed for the emerging hardware support. Surprisingly, the hardware VMM often suffers lower performance than the pure software VMM. To determine why, we study architecture-level events such as page table updates, context switches and I/O, and find their costs vastly different among native, software VMM and hardware VMM execution.

We find that the hardware support fails to provide an unambiguous performance advantage for two primary reasons: first, it offers no support for MMU virtualization; second, it fails to co-exist with existing software techniques for MMU virtualization. We look ahead to emerging techniques for addressing this MMU virtualization problem in the context of hardware-assisted 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, VT, SVM, MMU, TLB, Nested Paging

1. Introduction

The x86 has historically lacked hardware support for virtualization [21]. While paravirtualization [5, 25], or changing the guest operating system to permit virtualization, has produced promising results, such changes are not always practical or desirable.

The need to virtualize unmodified x86 operating systems has given rise to software techniques that go beyond the classical trap-and-emulate Virtual Machine Monitor (VMM). The best known of these software VMMs, VMware Workstation and Virtual PC, use binary translation to fully virtualize x86. The software VMMs have enabled widespread use of x86 virtual machines to offer server consolidation, fault containment, security and resource management.

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

Recently, the major x86 CPU manufacturers have announced architectural extensions to directly support virtualization in hardware. The transition from software-only VMMs to hardware-assisted VMMs provides an opportunity to examine the strengths and weaknesses of both techniques.

The main technical contributions of this paper are (1) a review of VMware Workstation's software VMM, focusing on performance properties of the virtual instruction execution engine; (2) a review of the emerging hardware support, identifying performance trade-offs; (3) a quantitative performance comparison of a software and a hardware VMM.

Surprisingly, we find that the first-generation hardware support rarely offers performance advantages over existing software techniques. We ascribe this situation to high VMM/guest transition costs and a rigid programming model that leaves little room for software flexibility in managing either the frequency or cost of these transitions.

While the first round of hardware support has been locked down, future rounds can still be influenced, and should be guided by an understanding of the trade-offs between today's software and hardware virtualization techniques. We hope our results encourage hardware designers to support the proven software techniques rather than seeking to replace them; we believe the benefits of software flexibility to virtual machine performance and functionality are compelling.

The rest of this paper is organized as follows. In Section 2, we review classical virtualization techniques and establish terminology. Section 3 describes our software VMM. Section 4 summarizes the hardware enhancements and describes how the software VMM was modified to exploit hardware support. Section 5 compares the two VMMs qualitatively and Section 6 presents experimental results and explains these in terms of the VMMs' properties. Section 7 looks ahead to future software and hardware solutions to the key MMU virtualization problem. Section 8 summarizes related work and Section 9 concludes.

2. Classical virtualization

Popek and Goldberg's 1974 paper [19] establishes three essential characteristics for system software to be considered a VMM:

1. *Fidelity.* Software on the VMM executes identically to its execution on hardware, barring timing effects.
2. *Performance.* An overwhelming majority of guest instructions are executed by the hardware without the intervention of the VMM.
3. *Safety.* The VMM manages all hardware resources.

In 1974, a particular VMM implementation style, trap-and-emulate, was so prevalent as to be considered the only practical

method for virtualization. Although Popek and Goldberg did not rule out use of other techniques, some confusion has resulted over the years from informally equating "virtualizability" with the ability to use trap-and-emulate.

To side-step this confusion we shall use the term *classically virtualizable* to describe an architecture that can be virtualized purely with trap-and-emulate. In this sense, x86 is not classically virtualizable, but it *is* virtualizable by Popek and Goldberg's criteria, using the techniques described in Section 3.

In this section, we review the most important ideas from classical VMM implementations: de-privileging, shadow structures and traces. Readers who are already familiar with these concepts may wish to skip forward to Section 3.

2.1 De-privileging

In a classically virtualizable architecture, all instructions that read or write privileged state can be made to trap when executed in an unprivileged context. Sometimes the traps result from the instruction type itself (e.g., an `out` instruction), and sometimes the traps result from the VMM protecting structures that the instructions access (e.g., the address range of a memory-mapped I/O device).

A *classical VMM* executes guest operating systems directly, but at a reduced privilege level. The VMM intercepts traps from the de-privileged guest, and emulates the trapping instruction against the virtual machine state. This technique has been extensively described in the literature (e.g., [10, 22, 23]), and it is easily verified that the resulting VMM meets the Popek and Goldberg criteria.

2.2 Primary and shadow structures

By definition, the privileged state of a virtual system differs from that of the underlying hardware. The VMM's basic function is to provide an execution environment that meets the guest's expectations in spite of this difference.

To accomplish this, the VMM derives *shadow structures* from guest-level *primary structures*. On-CPU privileged state, such as the page table pointer register or processor status register, is handled trivially: the VMM maintains an image of the guest register, and refers to that image in instruction emulation as guest operations trap.

However, off-CPU privileged data, such as page tables, may reside in memory. In this case, guest accesses to the privileged state may not naturally coincide with trapping instructions. For example, guest page table entries (PTEs) are privileged state due to their encoding of mappings and permissions. Dependencies on this privileged state are not accompanied by traps: every guest virtual memory reference depends on the permissions and mappings encoded in the corresponding PTE.

Such in-memory privileged state can be modified by any store in the guest instruction stream, or even implicitly modified as a side effect of a DMA I/O operation. Memory-mapped I/O devices present a similar difficulty: reads and writes to this privileged data can originate from almost any memory operation in the guest instruction stream.

2.3 Memory traces

To maintain coherency of shadow structures, VMMs typically use hardware page protection mechanisms to trap accesses to in-memory primary structures. For example, guest PTEs for which shadow PTEs have been constructed may be write-protected. Memory-mapped devices must generally be protected for both reading and writing. This page-protection technique is known as *tracing*. Classical VMMs handle a trace fault similarly to a privileged instruction fault: by decoding the faulting guest instruction, emulating its effect in the primary structure, and propagating the change to the shadow structure.

2.4 Tracing example: x86 page tables

To protect the host from guest memory accesses, VMMs typically construct *shadow page tables* in which to run the guest. x86 specifies hierarchical hardware-walked page tables having 2, 3 or 4 levels. The hardware page table pointer is control register `%cr3`.

VMware Workstation's VMM manages its shadow page tables as a cache of the guest page tables. As the guest accesses previously untouched regions of its virtual address space, hardware page faults vector control to the VMM. The VMM distinguishes *true page faults*, caused by violations of the protection policy encoded in the guest PTEs, from *hidden page faults*, caused by misses in the shadow page table. True faults are forwarded to the guest; hidden faults cause the VMM to construct an appropriate shadow PTE, and resume guest execution. The fault is "hidden" because it has no guest-visible effect.

The VMM uses traces to prevent its shadow PTEs from becoming incoherent with the guest PTEs. The resulting trace faults can themselves be a source of overhead, and other coherency mechanisms are possible. At the other extreme, avoiding all use of traces causes either a large number of hidden faults or an expensive context switch to prevalidate shadow page tables for the new context.

In our experience, striking a favorable balance in this three-way trade-off among trace costs, hidden page faults and context switch costs is surprising both in its difficulty and its criticality to VMM performance. Tools that make this trade-off more forgiving are rare and precious.

2.5 Refinements to classical virtualization

The type of workload significantly impacts the performance of the classical virtualization approach [20]. During the first virtual machine boom, it was common for the VMM, the hardware, and all guest operating systems to be produced by a single company. These vertically integrated companies enabled researchers and practitioners to refine classical virtualization using two orthogonal approaches.

One approach exploited flexibility in the VMM/guest OS interface. Implementors taking this approach modified guest operating systems to provide higher-level information to the VMM [13]. This approach relaxes Popek and Goldberg's fidelity requirement to provide gains in performance, and optionally to provide features beyond the bare baseline definition of virtualization, such as controlled VM-to-VM communication.

The other approach for refining classical VMMs exploited flexibility in the hardware/VMM interface. IBM's System 370 architecture introduced *interpretive execution* [17], a hardware execution mode for running guest operating systems. The VMM encodes much of the guest privileged state in a hardware-defined format, then executes the SIE instruction to "start interpretive execution." Many guest operations which would trap in a de-privileged environment directly access shadow fields in interpretive execution. While the VMM must still handle some traps, SIE was successful in reducing the frequency of traps relative to an unassisted trap-and-emulate VMM.

Both of these approaches have intellectual heirs in the present virtualization boom. The attempt to exploit flexibility in the OS/VMM layer has been revived under the name paravirtualization [25]. Meanwhile, x86 vendors are introducing hardware facilities inspired by interpretive execution; see Section 4.

3. Software virtualization

We review basic obstacles to classical virtualization of the x86 architecture, explain how binary translation (BT) overcomes the obstacles, and show that adaptive BT improves efficiency.

3.1 x86 obstacles to virtualization

Ignoring the legacy "real" and "virtual 8086" modes of x86, even the more recently architected 32- and 64-bit protected modes are not classically virtualizable:

- *Visibility of privileged state.* The guest can observe that it has been deprivileged when it reads its code segment selector (`%cs`) since the current privilege level (*CPL*) is stored in the low two bits of `%cs`.
- *Lack of traps when privileged instructions run at user-level.* For example, in privileged code `popf` ("pop flags") may change both ALU flags (e.g., ZF) and system flags (e.g., IF, which controls interrupt delivery). For a deprivileged guest, we need kernel mode `popf` to trap so that the VMM can emulate it against the virtual IF. Unfortunately, a deprivileged `popf`, like any user-mode `popf`, simply suppresses attempts to modify IF; no trap happens.

Other obstacles to classical virtualization exist on x86, but one obstacle is enough if it disrupts binary-only OS distributions like Windows.

3.2 Simple binary translation

The semantic obstacles to x86 virtualization can be overcome if the guest executes on an interpreter instead of directly on a physical CPU. The interpreter can prevent leakage of privileged state, such as the CPL, from the physical CPU into the guest computation and it can correctly implement non-trapping instructions like `popf` by referencing the virtual CPL regardless of the physical CPL. In essence, the interpreter separates virtual state (the VCPU) from physical state (the CPU).

However, while interpretation ensures Fidelity and Safety, it fails to meet Popek and Goldberg's Performance bar: the fetch-decode-execute cycle of the interpreter may burn hundreds of physical instructions per guest instruction. Binary translation, however, can combine the semantic precision of interpretation with high performance, yielding an execution engine that meets all of Popek and Goldberg's criteria. A VMM built around a suitable binary translator *can* virtualize the x86 architecture and it *is* a VMM according to Popek and Goldberg.

(We note in passing that use of BT for a VMM's execution engine has close parallels in other systems work: JVMs use JIT compilers [8]; architecture simulators and system emulators like Shade [7] and Embra [26] use translators to combine subject code and analysis code into fast target code.)

Our software VMM uses a translator with 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 telling code and data apart.
- *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 hex numbers in the resulting error message).
- *Subsetting.* The translator's input is the full x86 instruction set, including all privileged instructions; output is a safe subset (mostly user-mode instructions).
- *Adaptive.* Translated code is adjusted in response to guest behavior changes to improve overall efficiency.

To illustrate the translation process, we work through a small example. Since privileged instructions are rare even in OS kernels the performance of a BT system is largely determined by the translation of regular instructions, so our example is a simple primality test:

```
int isPrime(int a) {
  for (int i = 2; i < a; i++) {
    if (a % i == 0) return 0;
  }
  return 1;
}
```

We compiled the C code into this 64-bit binary:

```
isPrime:  mov    %ecx, %edi ; %ecx = %edi (a)
          mov    %esi, $2   ; i = 2
          cmp    %esi, %ecx ; is i >= a?
          jge    prime      ; jump if yes
nexti:    mov    %eax, %ecx ; set %eax = a
          cdq               ; sign-extend
          idiv   %esi       ; a % i
          test   %edx, %edx ; is remainder zero?
          jz     notPrime   ; jump if yes
          inc    %esi       ; i++
          cmp    %esi, %ecx ; is i >= a?
          jl     nexti      ; jump if no
prime:    mov    %eax, $1   ; return value in %eax
          ret
notPrime: xor    %eax, %eax ; %eax = 0
          ret
```

We invoked `isPrime(49)` 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:

```
89 f9 be 02 00 00 00 39 ce 7d ...
```

The translator reads the guest's memory at the address indicated by the guest PC, 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 it is rarely reached since control flow tends to terminate TUs sooner. Thus, in the common case a TU is a basic block (BB). The first TU in our example is:

```
isPrime:  mov %ecx, %edi
          mov %esi, $2
          cmp %esi, %ecx
          jge prime
```

Translating from x86 to x86 subset, most code can be translated IDENT (for "identically"). The first three instructions above are IDENT. `jge` must be non-IDENT since translation does not preserve code layout. Instead, we turn it into two translator-invoking continuations, one for each of the successors (fall-through and taken-branch), yielding this translation (square brackets indicate continuations):

```
isPrime':   mov %ecx, %edi   ; IDENT
            mov %esi, $2
            cmp %esi, %ecx
            jge [takenAddr]  ; JCC
            jmp [fallthrAddr]
```

Each translator invocation consumes one TU and produces one compiled code fragment (CCF). Although we show CCFs in textual form with labels like `isPrime'` to remind us that the address contains the translation of `isPrime`, in reality the translator produces binary code directly and tracks the input-to-output correspondence with a hash table.

Continuing our example, we now execute the translated code. Since we are calculating `isPrime(49)`, `jge` is not taken (`%ecx` is 49), so we proceed into the `fallthrAddr` case and invoke the translator on guest address `nexti`. This second TU ends with `jz`. Its translation is similar to the previous TU's translation with all but the final `jz` being IDENT.

To speed up inter-CCF transfers, the translator, like many previous ones [7], 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.

This interleaving of translation and execution continues for as long as the guest runs, with a decreasing proportion of translation as the TC gradually captures the guest's working set. For `isPrime`, after looping the `for` loop for long enough to detect that 49 isn't a prime, we end up with this code in the TC:

```
isPrime':  *mov   %ecx, %edi   ; IDENT
            mov   %esi, $2
            cmp   %esi, %ecx
            jge   [takenAddr]  ; JCC
                               ; fall-thru into next CCF
nexti':    *mov   %eax, %ecx   ; IDENT
            cdq
            idiv  %esi
            test  %edx, %edx
            jz    notPrime'    ; JCC
                               ; fall-thru into next CCF
           *inc   %esi         ; IDENT
            cmp   %esi, %ecx
            jl    nexti'       ; JCC
            jmp   [fallthrAddr3]

notPrime': *xor   %eax, %eax   ; IDENT
            pop   %r11         ; RET
            mov   %gs:0xff39eb8(%rip), %rcx  ; spill %rcx
            movzx %ecx, %r11b
            jmp   %gs:0xfc7dde0(8*%rcx)
```

Above, there are four CCFs with the leading instruction in each one marked with an asterisk. Two continuations remain because they were never executed while two disappeared entirely and one was replaced with a chaining jump to `nexti'`. For a bigger example than `isPrime`, but nevertheless one that runs in exactly the same manner, a 64-bit Windows XP Professional boot/halt translates 229,347 64-bit TUs, 23,909 32-bit TUs, and 6,680 16-bit TUs.

Since 49 isn't a prime number, we never translate the BB that returns 1 in `isPrime`. More generally, the translator captures an execution trace of the guest code, ensuring that TC code has good icache locality if the first and subsequent executions follow similar paths through guest code. Error-handling and other rarely executed guest code tends to get translated later than the first execution (if ever), causing placement away from the hot path.

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 are bound 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. The `ret` translation above uses a `%gs` override to spill `%rcx` into VMM memory so that it can be used as a working register in the translation of `ret`. Later, of course, the guest's `%rcx` value must be reloaded into the hardware `%rcx`.

`isPrime` is atypical in that it contains no memory accesses. However, memory accesses are common so their translation must run at close to native speed *and* have a form that prevents unintentional access to the VMM. The efficiency requirement favors use of hardware protection over insertion of explicit address checks.

x86 offers two protection mechanisms: paging and segmentation. For BT, segmentation works best. We map the VMM in the high part of the guest's address space and use segmentation to segregate guest portions (low) and VMM portions (high) of the address space. We then "truncate" guest segments so that they don't overlap the VMM. When all segment registers (but `%gs`) hold truncated segments, a fault ensues should a translated instruction attempt access to the VMM. Selectively, the translator inserts `%gs` prefixes to gain access to the VMM space. And, conversely, for the occasional guest instruction that has a `%gs` prefix, the translator strips it and uses a non-IDENT translation.

While most instructions can be translated IDENT, there are several noteworthy exceptions:

- *PC-relative addressing* cannot be translated IDENT since the translator output resides at a different address than the input. The translator inserts compensation code to ensure correct addressing. The net effect is a small code expansion and slowdown.
- *Direct control flow.* Since code layout changes during translation, control flow must be reconnected in the TC. For direct calls, branches and jumps, the translator can do the mapping from guest address to TC address. The net slowdown is insignificant.
- *Indirect control flow* (jmp, call, ret) does not go to a fixed target, preventing translation-time binding. Instead, the translated target must be computed dynamically, e.g., with a hash table lookup. The resulting overhead varies by workload but is typically a single-digit percentage.
- *Privileged instructions.* We use in-TC sequences for simple operations. These may run faster than native: e.g., `cli` (clear interrupts) on a Pentium 4 takes 60 cycles whereas the translation runs in a handful of cycles ("`vcpu.flags.IF:=0`"). Complex operations like context switches call out to the runtime, causing measurable overhead due both to the callout and the emulation work.

Finally, although the details are beyond the scope of this paper, we observe that BT is not required for safe execution of most user code on most guest operating systems. By switching guest execution between BT mode and *direct execution* as the guest switches between kernel- and user-mode, we can limit BT overheads to kernel code and permit application code to run at native speed.

3.3 Adaptive binary translation

Modern CPUs have expensive traps, so a BT VMM can outperform a classical VMM by avoiding privileged instruction traps. To illustrate, we compared implementations of a simple privileged instruction (`rdtsc`) on a Pentium 4 CPU: trap-and-emulate takes 2030 cycles, callout-and-emulate takes 1254 cycles, and in-TC emulation takes 216 cycles.

However, while simple BT eliminates traps from privileged instructions, an even more frequent trap source remains: non-

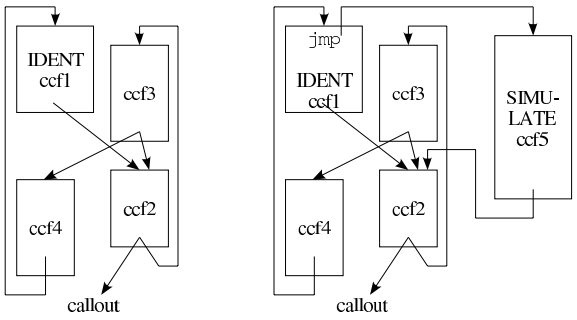

Figure 1. Adaptation from IDENT to SIMULATE.

privileged instructions (e.g., loads and stores) accessing sensitive data such as page tables. We use *adaptive BT* to essentially eliminate the latter category of traps. The basic idea is "innocent until proven guilty." Guest instructions start in the innocent state, ensuring maximal use of IDENT translations. During execution of translated code we detect instructions that trap frequently and adapt their translation:

- Retranslate non-IDENT to avoid the trap; for example, the translation may call out to an interpreter.
- Patch the original IDENT translation with a forwarding jump to the new translation.

The left half of Figure 1 shows a control flow graph with an IDENT translation in ccf1, and arbitrary other control flow in the TC represented by ccf2, ccf3 and ccf4. The right half shows the result of adapting from IDENT in ccf1 to translation type SIMULATE in ccf5. Adaptation takes constant time since use of a forwarding jump within ccf1 avoids the need to visit all ccf1's direct ancestors to redirect exit control flow from ccf1 to ccf5.

After adaptation, we avoid taking a trap in ccf1 and instead execute a faster callout in ccf5. The simulate callout continues to monitor the behavior of the offending instruction. If the behavior changes and the instruction becomes innocent again, we switch the active translation type back to IDENT by removing the forwarding jump from ccf1 and inserting an opposing one in ccf5.

The VMM uses adaptation not just in a bimodal form that distinguishes between innocent and guilty instructions, but with the ability to adapt to a variety of situations, including access to a page table, access to a particular device, and access to the VMM's address range.

A guest instruction whose translation has been adapted suffers a dynamic overhead of a forwarding jump to reach the replacement translation. Adaptation's static overhead, i.e., code patching and resulting loss of icache contents, can be controlled with hysteresis to ensure a low adaptation frequency. We adapt aggressively from a trapping translation to a non-trapping more general form, but less aggressively towards a more optimistic translation.

4. Hardware virtualization

In this section, we discuss recent architectural changes that permit classical virtualization of the x86. The discussion applies to both AMD's SVM and Intel's VT; the similarity between the two architectures is obvious from their respective manuals [2, 12]. VMware has implemented an experimental VMM to exploit these new hardware capabilities. We describe this hardware-assisted VMM (for brevity: hardware VMM), and compare it with the software VMM.

4.1 x86 architecture extensions

The hardware exports a number of new primitives to support a classical VMM for the x86. An in-memory data structure, which we will refer to as the *virtual machine control block*, or VMCB, combines control state with a subset of the state of a guest virtual CPU. A new, less privileged execution mode, *guest mode*, supports direct execution of guest code, including privileged code. We refer to the previously architected x86 execution environment 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 control bits of the VMCB, is reached. At this point, the hardware performs an *exit* operation, which is the inverse of a `vmrun` operation. On exit, the hardware saves guest state to the VMCB, loads VMM-supplied state into the hardware, and resumes in host mode, now executing the VMM.

Diagnostic fields in the VMCB aid the VMM in handling the exit; e.g., exits due to guest I/O provide the port, width, and direction of I/O operation. After emulating the effect of the exiting operation in the VMCB, the VMM again executes `vmrun`, returning to guest mode.

The VMCB control bits provide some flexibility in the level of trust placed in the guest. For instance, a VMM behaving as a hypervisor for a general-purpose OS might allow that OS to drive system peripherals, handle interrupts, or build page tables. However, when applying hardware assistance to pure virtualization, the guest must run on a shorter leash. The hardware VMM programs the VMCB to exit on guest page faults, TLB flushes, and address-space switches in order to maintain the shadow page tables; on I/O instructions to run emulated models of guest peripherals; and on accesses to privileged data structures such as page tables and memory-mapped devices.

4.2 Hardware VMM implementation

The hardware extensions provide a complete virtualization solution, essentially prescribing the structure of our hardware VMM (or indeed any VMM using the extensions). When running a protected mode guest, the VMM fills in a VMCB with the current guest state and executes `vmrun`. On guest exits, the VMM reads the VMCB fields describing the conditions for the exit, and vectors to appropriate emulation code.

Most of this emulation code is shared with the software VMM. It includes peripheral device models, code for delivery of guest interrupts, and many infrastructure tasks such as logging, synchronization and interaction with the host OS. Since current virtualization hardware does not include explicit support for MMU virtualization, the hardware VMM also inherits the software VMM's implementation of the shadowing technique described in Section 2.

4.3 Example operation: process creation

In explaining the BT-based VMM, we used `isPrime` as an example guest program. On a hardware VMM, `isPrime` is uninteresting, because, containing only ALU operations and control-flow, its execution is identical in host and guest mode. We need a more substantial operation to illustrate the operation of a hardware VMM. So, consider a UNIX-like operating system running in guest mode on the hardware VMM, about to create a process using the `fork(2)` system call.

- A user-level process invokes `fork()`. The system call changes the CPL from 3 to 0. Since the guest's trap and system call vectors are loaded onto the hardware, the transition happens without VMM intervention.

- In implementing fork, the guest uses the "copy-on-write" approach of write-protecting both parent and child address spaces. Our VMM's software MMU has already created shadow page tables for the parent address space, using traces to maintain their coherency. Thus, each guest page table write causes an exit. The VMM decodes the exiting instruction to emulate its effect on the traced page and to reflect this effect into the shadow page table. By updating the shadow page table, the VMM write-protects the parent address space in the hardware MMU.
- The guest scheduler discovers that the child process is runnable and context switches to it. It loads the child's page table pointer, causing an exit. The VMM's software MMU constructs a new shadow page table and points the VMCB's page table register at it.
- As the child runs, it touches pieces of its address space that are not yet mapped in its shadow page tables. This causes hidden page fault exits. The VMM intercepts the page faults, updates its shadow page table, and resumes guest execution.
- As both the parent and child run, they write to memory locations, again causing page faults. These faults are true page faults that reflect protection constraints imposed by the guest. The VMM must still intercept them before forwarding them to the guest, to ascertain that they are not an artifact of the shadowing algorithm.

4.4 Discussion

The VT and SVM extensions make classical virtualization possible on x86. The resulting performance depends primarily on the frequency of exits. A guest that never exits runs at native speed, incurring near zero overhead. However, this guest would not be very useful since it can perform no I/O. If, on the other hand, every instruction in the guest triggers an exit, execution time will be dominated by hardware transitions between guest and host modes. *Reducing the frequency of exits is the most important optimization for classical VMMs.*

To help avoid the most frequent exits, x86 hardware assistance includes ideas similar to the s/370 interpretive execution facility discussed in Section 2.5. Where possible, privileged instructions affect state within the virtual CPU as represented within the VMCB, rather than unconditionally trapping [24].

Consider again `popf`. A naive extension of x86 to support classical 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 as shown with the fork example above.

5. Qualitative comparison

An ideal VMM runs the guest at native speed. The software and hardware VMMs experience different trade-offs in their attempts to approach this ideal. BT tends to win in these areas:

- *Trap elimination:* adaptive BT can replace most traps with faster callouts.
- *Emulation speed:* a callout can provide the emulation routine with a predecoded guest instruction, whereas a hardware VMM must fetch and decode the trapping guest instruction to emulate it.
- *Callout avoidance:* for frequent cases, BT may use in-TC emulation routines, avoiding even the callout cost.

Conversely, the hardware VMM wins in these areas:

- *Code density* is preserved since there is no translation.
- *Precise exceptions:* BT performs extra work to recover guest state for faults and interrupts in non-IDENT code.
- *System calls* run without VMM intervention.

In summary, hardware and software VMMs suffer different overheads. While software virtualization requires careful engineering to ensure efficient execution of guest kernel code, hardware virtualization delivers native speed for anything that avoids an exit but levies a higher cost for the remaining exits (on current hardware). The software VMM has a richer set of options available, including adaptation, whereas current hardware mechanisms aim more narrowly at trap-and-emulate style virtualization, leaving less flexibility to use other software/hardware combinations.

6. Experiments

We have examined a number of 64-bit workloads under VMware Player 1.0.1's software and hardware-assisted VMMs. Our hardware host is an HP xw4300 workstation, containing a VT-enabled 3.8 GHz Intel Pentium 4 672 with hyperthreading disabled in the BIOS.

Current Intel CPUs lack support for segment limits in 64 bit mode, leaving us without our preferred method for protecting the software VMM. So we caution that in the measured hardware environment the software VMM fails Popek and Goldberg's Safety requirement. However, the VMM exhibits similar performance on AMD's Opteron processor where 64-bit segment limits are supported, so we are confident that these measurements accurately represent the performance of the software VMM.

We qualify our results in this section by noting that we are comparing a recently developed hardware VMM on a particular microarchitecture with a mature commercial product. While the comparison cannot be completely "apples-to-apples," we feel it still offers insight. Improvements in software and hardware may change the constants, but the broad profile of the two approaches' strengths and weaknesses will remain the same.

We find that compute-intensive benchmarks run essentially at native speed on both VMMs. However, as workloads include progressively more privileged operations (context switches, memory mapping, I/O, interrupts, system calls), both VMMs suffer overheads. Using a series of increasingly targeted benchmarks we show how and why the software VMM usually outperforms the hardware VMM.

6.1 Initial measurements

As an example of a benign workload, we ran SPECint 2000 on Red-Hat Enterprise Linux 3. Since user-level computation is not taxing for VMMs, we expect both guest runs to score close to native. Figure 2 confirms this expectation, showing a slowdown over native of 0-9%, with a 4% average slowdown for the software VMM and 5% for the hardware VMM. The overhead results from both host background activity, such as timer interrupts and housekeeping kernel threads, and virtualization overheads in guest background activity. Surprisingly, `mcf` runs faster than native on both VMMs. Measurement with an external timer to rule out virtual timing jitter showed

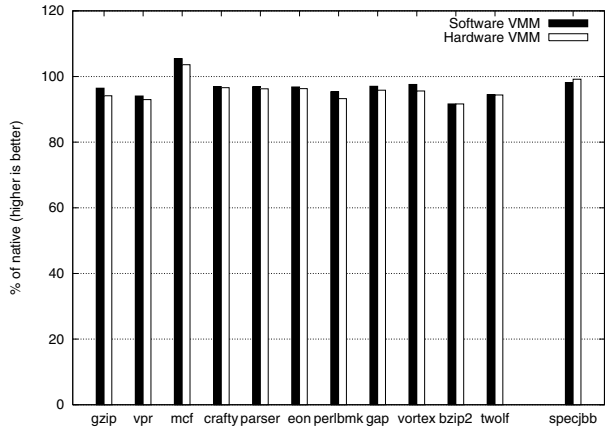

Figure 2. SPECint 2000 and SPECjbb 2005.

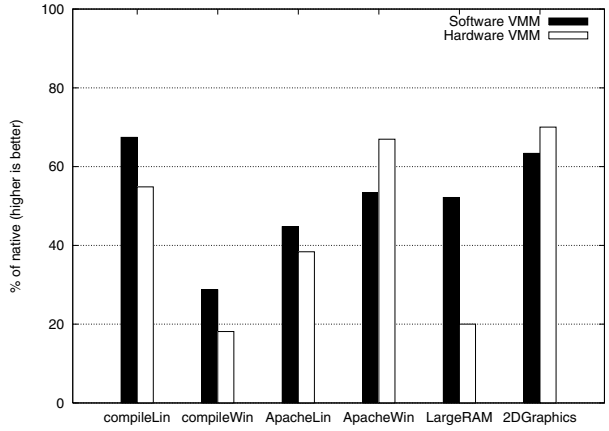

Figure 3. Macrobenchmarks.

the performance improvement to be real. We speculate that the VMM positively perturbs TLBs and caches for this combination of workload and hardware but have been unable to prove this conclusively because hardware performance counters cannot be applied to workloads running in VMware Player 1.0.1.

The SPECjbb 2005 benchmark tests server-side Java Virtual Machine (JVM) performance. To achieve guest variety, we ran SPECjbb on Windows 2003 Enterprise x64 Edition, using Sun JVM version 1.5.0_06-b05. The results in Figure 2, again normalized to native, show that both the software and hardware VMMs come very close to native performance, reaching 98% and 99%, respectively. Since the JVM runs as a single user-level process, direct execution dominates SPECjbb's runtime within a VM, just as with SPECint. This test came even closer to native performance than SPECint perhaps due to Windows 2003's lower timer interrupt rate (60Hz, vs. 100Hz for the Linux VM).

For a more challenging server workload, we ran the Apache `ab` server benchmarking tool with 128 clients against Linux and Windows installations of the Apache http server; see Figure 3.

VMware Player uses a hosted I/O model in which all network packets pass through the host OS's I/O stack. Due to this overhead, all four guest/VMM combinations compare poorly to native. However, the relative merits of the two VMMs are guest-dependent. On Windows, the hardware VMM achieves a superior ratio to native performance (67%) to that of the software VMM (53%). On Linux, however, the software VMM (45%) dominates the hardware VMM (38%). For reasons described below, we attribute this to differences in Apache configuration. Apache defaults to a single address space on Windows, and many address spaces in UNIX-like operating systems such as Linux.

For a desktop-oriented workload, we ran PassMark on Windows XP Professional x64 Edition both natively and in a VM. PassMark is a synthetic suite of microbenchmarks intended to isolate various aspects of workstation performance. We found that the two VMMs encountered similar overhead in most PassMark components. Many of these benchmarks stress I/O devices whose latencies easily hide differences in CPU virtualization.

However, two benchmarks revealed discrepancies between the VMMs, as shown in Figure 3. The "Large RAM" component exhausts the 1GB of RAM available in both host and guest, leading to paging. Both VMMs show a significant deficit relative to the native score of 335 op/s. However, the software VMM, scoring 175 op/s, fares much better than the hardware VMM at 67 op/s.

In the 2D graphics score, the host scored 590 op/s, while the software and hardware VMMs scored 374 op/s and 413 op/s respectively. In this case, the hardware VMM achieved 70% of native performance, while the software VMM achieved only 63%. Instrumentation of the software VMM confirmed that the 2D graphics workload suffers from system call overheads. We attribute the hardware VMM's success at this workload to its superiority in handling kernel/user transitions.

For a less synthetic workload, we examined compile jobs on both Linux ("make -j8 bzImage" for kernel 2.6.11) and Windows (Cygwin "make" Apache 2.2.0). We see a similar pattern for both guests with the software VMM being closer to native. In the Linux compile job, the host took 265 seconds, the software VMM took 393 seconds (67.4% of native performance), and the hardware VMM took 484 seconds (54.8% of native). Put differently, the software VMM's overhead is 128 seconds whereas the hardware VMM's overhead is almost twice as large at 219 seconds. The Windows compile job behaves similarly, although both VMMs display a larger gap relative to native due to IPC overheads (i.e., additional context switches) from the Cygwin UNIX emulation environment.

Investigating the hardware VMM's large deficit in kernel compilation, we gathered samples of the guest program counter while executing under both VMMs. We found that the guest spends more time servicing page faults and context switches on the hardware VMM. Our next benchmark zooms in on these overheads.

6.2 Forkwait

To magnify the differences between the two VMMs, we use the familiar UNIX kernel microbenchmark `forkwait`, which stresses process creation and destruction. The program is perhaps most concisely described by its source:

```
int main(int argc, char *argv[]) {
  for (int i = 0; i < 40000; i++) {
    int pid = fork();
    if (pid < 0)  return -1;
    if (pid == 0) return 0;
    waitpid(pid);
  }
  return 0;
}
```

`forkwait` focuses intensely on virtualization-sensitive operations, resulting in low performance relative to native execution. Measuring `forkwait`, our host required 6.0 seconds to create and destroy 40000 processes. The software VMM, on the other hand, took 36.9 seconds, while the hardware VMM consumed a sobering 106.4 seconds. `forkwait` effectively magnifies the difference be-

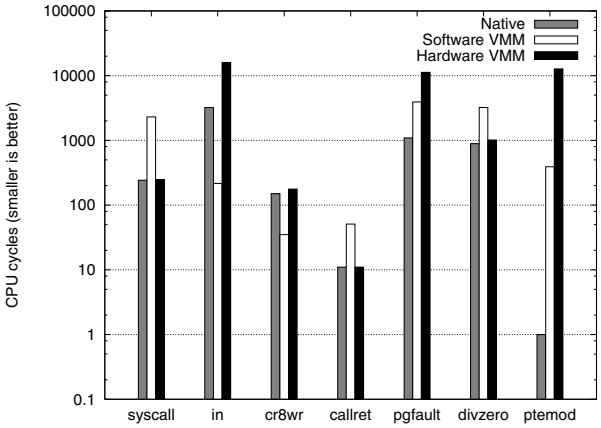

Figure 4. Virtualization nanobenchmarks.

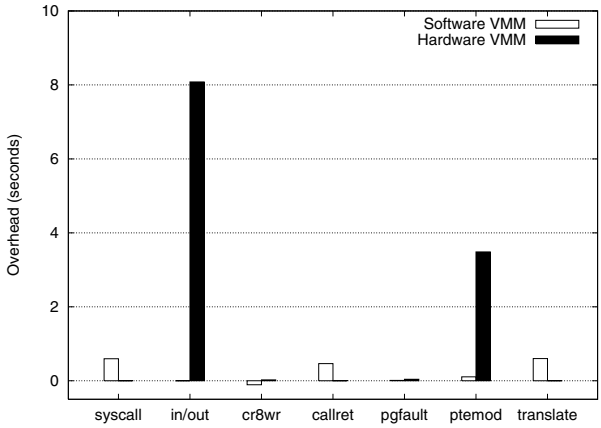

Figure 5. Sources of virtualization overhead in an XP boot/halt.

tween the two VMMs, the hardware VMM inducing approximately 4.4 times greater overhead than the software VMM. Still, this program stresses many divergent paths through both VMMs, such as system calls, context switching, creation of address spaces, modification of traced page table entries, and injection of page faults.

6.3 Virtualization nanobenchmarks

To better understand the performance differences between the two VMMs, we wrote a series of "nanobenchmarks" that each exercise a single virtualization-sensitive operation. Often, the measured operation is a single instruction long. For precise control over the executed code, we repurposed a custom OS, FrobOS, that VMware developed for VMM testing.

Our modified FrobOS boots, establishes a minimal runtime environment for C code, calibrates its measurement loops, and then executes a series of virtualization-sensitive operations. The test repeats each operation many times, amortizing the cost of the binary translator's adaptations over multiple iterations. In our experience, this is representative of guest behavior, in which adaptation converges on a small fraction of poorly behaving guest instructions. The results of these nanobenchmarks are presented in Figure 4. The large spread of cycle counts requires the use of a logarithmic scale.

`syscall`. This test measures round-trip transitions from user-level to supervisor-level via the `syscall` and `sysret` instructions. The software VMM introduces a layer of code and an extra privilege transition, requiring approximately 2000 more cycles than a native system call. In the hardware VMM, system calls execute without VMM intervention, so as we expect, the hardware VMM executes system calls at native speed.

`in`. We execute an `in` instruction from port 0x80, the BIOS POST port. Native execution accesses an off-CPU register in the chipset, requiring 3209 cycles. The software VMM, on the other hand, translates the `in` into a short sequence of instructions that interacts with the virtual chipset model. Thus, the software VMM executes this instruction fifteen times faster than native. The hardware VMM must perform a vmm/guest round trip to complete the I/O operation. This transition causes `in` to consume 15826 cycles in the tested system.

`cr8wr`. `%cr8` is a privileged register that determines which pending interrupts can be delivered. Only `%cr8` writes that reduce `%cr8` below the priority of the highest pending virtual interrupt cause an exit [24]. Our FrobOS test never takes interrupts so no `%cr8` write in the test ever causes an exit. As with `syscall`, the hardware VMM's performance is similar to native. The software VMM translates `%cr8` writes into a short sequence of simple instructions, completing the `%cr8` write in 35 cycles, about four times faster than native.

`call/ret`. BT slows down indirect control flow. We target this overhead by repeatedly calling a subroutine. Since the hardware VMM executes calls and returns without modification, the hardware VMM and native both execute the call/return pair in 11 cycles. The software VMM introduces an average penalty of 40 cycles, requiring 51 cycles.

`pgfault`. In both VMMs, the software MMU interposes on both true and hidden page faults. This test targets the overheads for true page faults. While both VMM paths are logically similar, the software VMM (3927 cycles) performs much better than the hardware VMM (11242 cycles). This is due mostly to the shorter path whereby the software VMM receives control; page faults, while by no means cheap natively (1093 cycles on this hardware), are faster than a `vmrun`/exit round-trip.

`divzero`. Division by zero has fault semantics similar to those of page faults, but does not invoke the software MMU. While division by zero is uncommon in guest workloads, we include this nanobenchmark to clarify the `pgfault` results. It allows us to separate out the virtualization overheads caused by faults from the overheads introduced by the virtual MMU. As expected, the hardware VMM (1014 cycles) delivers near native performance (889 cycles), decisively beating the software VMM (3223 cycles).

`ptemod`. Both VMMs use the shadowing technique described in Section 2.4 to implement guest paging with trace-based coherency. The traces induce significant overheads for PTE writes, causing very high penalties relative to the native *single* cycle store. The software VMM adaptively discovers the PTE write and translates it into a small program that is cheaper than a trap but still quite costly. This small program consumes 391 cycles on each iteration. The hardware VMM enters and exits guest mode repeatedly, causing it to perform approximately thirty times worse than the software VMM, requiring 12733 cycles.

To place this data in context, Figure 5 shows the total overheads incurred by each nano-operation during a 64-bit Windows XP Professional boot/halt. Although the `pgfault` nanobenchmark has much higher cost on the hardware VMM than the software VMM, the boot/halt workload took so few true page faults that the difference does not affect the bottom line materially. In contrast, the guest performed over 1 million PTE modifications, causing high overheads for the hardware VMM. While the figure may suggest that `in/out` dominates the execution profile of the hardware VMM, the vast majority of these instructions originate in atypical BIOS code that is unused after initial boot.

	3.8GHz P4 672	2.66GHz Core 2 Duo
VM entry	2409	937
Page fault VM exit	1931	1186
VMCB read	178	52
VMCB write	171	44

Table 1. Micro-architectural improvements (cycles).

System calls were similar in frequency to PTE modifications. However, while the software VMM slows down system calls substantially, on an end-to-end basis system calls were not frequent enough to offset the hardware VMM's penalty for PTE modification (and I/O instructions), and the hardware VMM incurs considerably more total overhead than the software VMM in this workload.

The cost of running the binary translator (vs. executing the translated code) is rarely significant; see again Figure 5. There are two reasons. First, the TC captures the working set and continued execution amortizes away translation costs for long-running workloads. Second, the translator is quite fast because it does little analysis (2300 cycles per x86 instruction, compared with 100-200 kcycles per Java bytecode for some optimizing JITs [1]). High translator throughput ensures good performance even for a worst-case workload like boot/halt that mostly executes cold code.

In Section 2.4, we discussed the importance of finding a sweet spot in the three-way trade-off among trace costs, hidden page faults and context-switch costs. These nanobenchmarks demonstrate that hardware virtualization makes all three dimensions of this design space more treacherous by replacing faults with costlier guest/vmm round-trips. Further, in the absence of a hardware basis for supporting BT execution the hardware VMM is unable to adapt to frequently exiting workloads. Without this tool for easing the three-way trade-off, we have found few workloads that benefit from current hardware extensions.

We discovered two such workloads. Apache on Windows and 2D PassMark are similar in that they consist of a single address space, and the workloads require few I/O exits relative to the number of system calls performed. The few I/O exits are a consequence of guest buffering, in the case of Apache, and of the virtual SVGA device (which uses a shared-memory FIFO of graphics operations, usually avoiding exits until the FIFO fills up) in the case of PassMark. We consider these workloads the exception that proves the rule, however. Workloads that enter CPL 0 frequently, but rarely perform privileged operations are unusual. Recall the old UNIX benchmarking cheat of returning a cached value from `getpid(2)`: if the application isn't asking for a privileged service, why enter the kernel at all?

7. Software and hardware opportunities

Many of the difficult cases for the hardware VMM examined in Section 6.3 surround MMU virtualization. In this section, we consider future approaches in both hardware and software to close the gap to software VMM performance, and ultimately approach native performance.

7.1 Microarchitecture

Hardware overheads will shrink over time as implementations mature. Measurements on a desktop system using a pre-production version of Intel's next generation "Core" microarchitecture to ship in the second half of 2006 demonstrates that this positive trend is already under way; see Table 1. The number of cycles for a VM entry drops from 2409 on P4 to 937 on Core, a 61% reduction. Factoring in Core's lower clock, the time reduction is still a respectable 44% from 634 ns to 352 ns. Core offers a less dramatic 12% improvement in the cost of a page fault VM exit: from 508 ns to 446 ns.

This microarchitectural improvement visibly impacts the bottom line. Let us illustrate by comparing `forkwait` on P4 and Core. Native execution of `forkwait` on our P4 takes 6.02 seconds whereas our Core CPU runs it in just 2.62 seconds. This dramatic difference in native performance means that it is not meaningful to directly compare execution times of the hardware VMMs. Instead, we compare ratios to native. On P4, the hardware VMM runs `forkwait` 17.7x slower than native in 106.4 seconds. On Core, the hardware VMM improves the score to 14.8x slower than native, taking 38.9 seconds. For reference, the software VMM is 6.1x slower than native on P4 (36.9 seconds) and 7.0x slower than native on Core (18.4 seconds). Keeping in mind the extreme nature of `forkwait`, the improvement from 17.7x to 14.8x is significant, but it is also clear that more work remains to be done before virtual performance approaches native.

While this hardware VMM performance improvement is encouraging, we speculate that even a hypothetical "perfect" microarchitecture in which VM exits are free could have a performance deficit relative to the software VMM for MMU-related operations. The remaining performance gap is due to the "stateless" nature of the hardware VMM: after resuming a guest in direct hardware-assisted execution, the VMM has little idea what state the guest is in when execution returns to the VMM. So the VMM incurs software overheads reconstructing guest state by reading VMCB fields (handling a typical exit requires ten `vmread`s) and in some cases decoding the exiting instruction. While improvements on this state reconstruction software are certainly possible, a complete elimination of it is unlikely. "Stateless" VMM operation is characteristic of hardware-assisted direct execution. Thus, the opportunities for making exits faster, in both hardware and software, are limited.

7.2 Hardware VMM algorithmic changes

A more potent approach is to eliminate exits entirely. Consider, for instance, the `ptemod` nanobenchmark, which repeatedly modifies a guest page table entry. VMware Player 1.0.1 incurs a VM exit for each modification due to traces on shadow page tables. However, the x86's relaxed TLB coherency rules permit other page table coherency strategies than tracing. For example, VMware Workstation 2.0, released in 2000, would aggressively drop traces upon guest page table modification, allowing the shadow page tables to become temporarily incoherent with the guest page tables in order to reduce trace costs.

While this algorithm reduces trace costs, it shifts work into the VMM's guest page table reload path, which must process the backlog of page table writes to bring the shadow and guest page tables into harmony. The algorithm also prevents some non-obvious optimizations that are possible with more aggressive tracing. For instance, page table modifications that change a page from not-present to present usually occur in guest kernels' page fault handlers in response to demand-paging activity. Thus, these PTE modifications are almost always followed immediately by a user-level access to the freshly mapped page. By catching PTE writes with tracing, we can save a subsequent hidden page fault by eagerly placing a corresponding present entry in the shadow as soon as the guest PTE write executes. This optimization is important in process creation workloads such as `forkwait`, and is hard to apply without tracing.

In the years since the release of Workstation 2.0, VMware's binary translator has become increasingly capable of adapting to trace accesses, allowing our software MMU to make liberal use of traces. This extreme point in the design space is not optimal for the hardware-assisted VMM where trace accesses are costlier. Consequently, an effort to explore the design space of less trace-intensive software MMU algorithms is now under way at VMware. While this work is still in progress, and beyond the scope of this

paper, preliminary results appear promising: the hardware VMM gains on the software VMM, but still does not surpass it.

7.3 A hybrid VMM

For clarity of comparison, we have discussed the software and hardware VMM as entirely separate entities. However, both VMMs are part of the same binary in VMware Player 1.0.1, and nothing forces us to rely exclusively on one technique for executing a guest. We have experimented with a hybrid VMM that dynamically selects the execution technique using simple guest behavior-driven heuristics, with the goal of reaping the benefits of both the hardware VMM's superior system call performance and the software VMM's superior MMU performance. While we achieved some encouraging improvements on toy workloads, the heuristics driving this adaptation are still immature and require further tuning. For production use of BT on Intel CPUs we would additionally need to implement a performant VMM protection scheme that doesn't require segmentation.

7.4 Hardware MMU support

We are optimistic about the potential of future hardware assistance in the area of MMU virtualization. The three-way trade-off among trace costs, hidden page faults and context-switch costs was successfully addressed with appropriate hardware in IBM's s/370 machines [17]. Both AMD's "nested paging" [2] and Intel's "EPT" [16] proposals for MMU virtualization are similar to the SIE design.

In both schemes, the VMM maintains a hardware-walked "nested page table" that translates guest physical addresses to host physical addresses. 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 6. While running in hardware-assisted guest execution, the TLB contains entries mapping guest virtual addresses all the way to host physical addresses. The process of filling the TLB in case of a miss is somewhat more complicated than that of typical virtual memory systems. 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. `%cr3` contains a guest physical address, which must be translated to a host physical address before dereferencing. The hardware walks the nested page table for the guest's `%cr3` value to obtain a host physical pointer to the top level of the guest's page table hierarchy.
3. The hardware reads the guest page directory entry corresponding to guest virtual address V.
4. The PDE read in step 3 also yields a guest physical address, which must also be translated via the nested page table before proceeding.
5. Having discovered the host physical address 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 PTE points to guest physical address X, which is translated via a third walk of the nested page table, e.g. to host physical address Y.
6. The translation is complete: virtual address V maps to host physical address Y. The page walk hardware can now fill the TLB with an appropriate entry (V, Y) and resume guest execution, all without software intervention.

For an M-level guest page table on an N-level nested page table, a worst-case TLB miss requires MN memory accesses to satisfy. We are, however, optimistic about this potential problem. The same microarchitectural implementation techniques that make virtual memory perform acceptably (highly associative, large, multi-level TLBs, caching) should apply at least as well to the nested page table. Thus, nested paging holds the promise of eliminating trace overheads and allowing guest context switches without VMM intervention. By resolving the most important sources of overhead in current VMMs, nested paging hardware should easily repay the costs of (slightly) slower TLB misses.

8. Related work

Our approach to this topic owes a debt to the long-running RISC vs. CISC debate [6, 18]. This quarrel in its narrow sense has been fought to a draw, with a current abundance of both RISC and CISC designs. However, the controversy's lasting gifts have been skepticism towards the intuition that hardware always outperforms software, and the consensus that measurement must guide design of the hardware/software interface [11].

The picoJava microprocessor lands at one end of the hardware/software split [15]. This processor effectively reimplements the software JVM version 1.0 in hardware. While picoJava was never intended as a high performance bytecode execution engine, it is still striking to observe just how differently a modern JVM delivers performance through multi-stage JIT compilation. The picoJava design, frozen in time in the last millenium, is not what one would build today to run Java.

Transmeta's Crusoe design team took the opposite approach, building an x86 compatible CPU by codesigning a VLIW core and a substantial software component that includes an optimizing, dynamic, adaptive, binary translator from x86 to the internal VLIW architecture [9]. Their core implements basic primitives to support key aspects of the software, including speculation and self-modifying code.

The emergence of open source operating systems, such as Linux, has enabled paravirtualization, the modification of operating systems to be better virtual citizens [5, 25, 14]. The approach has merit as it gains performance and permits a simpler VMM design. However, it sacrifices the ability to run legacy and proprietary operating systems. Moreover, a new interface between the guest and VMM must now be standardized, documented and supported in addition to the existing interface between OS and hardware [3].

9. Conclusions

We have described the implementation of a software VMM that employs BT for x86 guest execution. Recent hardware extensions now permit implementation of a trap-and-emulate hardware VMM that executes guests directly.

Intuitively, one would expect hardware support to improve performance across the board, but experiments on first generation hardware painted a mixed picture. Our software and hardware VMMs both perform well on compute-bound workloads. For workloads that perform I/O, create processes, or switch contexts rapidly, software outperforms hardware. In two workloads rich in system calls, the hardware VMM prevails.

We have traced this surprising result to its root cause by studying architecture-level operations for which the software and hardware VMMs impose very different amounts of overhead. While the new hardware removes the need for BT and simplifies VMM design, in our experiments it rarely improves performance. New MMU algorithms in the hardware VMM might narrow this gap, but current hardware support is CISC-like in its packaging of a whole solution and difficult to exploit without giving up existing software techniques.

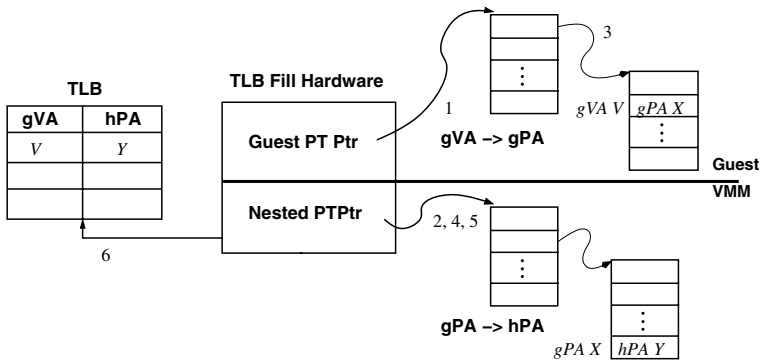

Figure 6. Nested paging hardware.

Over years of work on VMMs, we have come to appreciate the flexibility of having software control in our VMMs. We hope that coming hardware support for virtualization will be designed to blend easily with and complement existing software techniques. This may contribute to faster adoption of the hardware and advance the general state of virtualization.

Acknowledgments. Special thanks to Jim Mattson for significant contributions to the hardware VMM. General thanks to the dozens of members of VMware's VMM group since 1998. Warmest thanks to our readers: Eli Collins, Erik Cota-Robles, Alex Garthwaite, Geoff Pike, Mendel Rosenblum, and Richard Uhlig. We would also like to thank the anonymous ASPLOS reviewers for suggestions that greatly shaped our paper.

References

[1] AGESEN, O., AND DETLEFS, D. Mixed-mode bytecode execution. Technical Report SMLI TR-200-87, Sun Microsystems, Inc., Mountain View, CA, USA, 2000.

[2] AMD. *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*, May 2005.

[3] AMSDEN, Z., ARAI, D., HECHT, D., HOLLER, A., AND SUBRAHMANYAM, P. VMI: An interface for paravirtualization. *Ottawa Linux Symposium* (2006).

[4] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on programming language design and implementation* (New York, NY, USA, 2000), ACM Press, pp. 1–12.

[5] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on operating systems principles* (New York, NY, USA, 2003), ACM Press, pp. 164–177.

[6] CLARK, D. W., AND STRECKER, W. D. Comments on "the case for the reduced instruction set computer," by Patterson and Ditzel. *SIGARCH Comput. Archit. News 8*, 6 (1980), 34–38.

[7] CMELIK, B., AND KEPPEL, D. 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* (New York, NY, USA, 1994), ACM Press, pp. 128–137.

[8] CRAMER, T., FRIEDMAN, R., MILLER, T., SEBERGER, D., WILSON, R., AND WOLCZKO, M. Compiling java just in time. *IEEE Micro 17*, 3 (1997), 36–43.

[9] DEHNERT, J. C., GRANT, B. K., BANNING, J. P., JOHNSON, R., KISTLER, T., KLAIBER, A., AND MATTSON, J. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on code generation and optimization* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 15–24.

[10] DEITEL, H. M. *An introduction to operating systems (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[11] HENNESSY, J. L., AND PATTERSON, D. A. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[12] INTEL CORPORATION. *Intel® Virtualization Technology Specification for the IA-32 Intel® Architecture*, April 2005.

[13] KARGER, P. A., ZURKO, M. E., BONIN, D. W., MASON, A. H., AND KAHN, C. E. A vmm security kernel for the vax architecture. In *IEEE Symposium on Security and Privacy* (1990), pp. 2–19.

[14] LEVASSEUR, J., UHLIG, V., CHAPMAN, M., CHUBB, P., LESLIE, B., AND HEISER, G. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), Nov. 2005.

[15] MCGHAN, H., AND O'CONNOR, M. Picojava: A direct execution engine for java bytecode. *Computer 31*, 10 (1998), 22–30.

[16] NEIGER, G., SANTONI, A., LEUNG, F., RODGERS, D., AND UHLIG, R. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal 10*, 3 (2006).

[17] OSISEK, D. L., JACKSON, K. M., AND GUM, P. H. ESA/390 interpretive-execution architecture, foundation for VM/ESA. *IBM Systems Journal 30*, 1 (1991), 34–51.

[18] PATTERSON, D. A., AND DITZEL, D. R. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News 8*, 6 (1980), 25–33.

[19] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (1974), 412–421.

[20] POPEK, G. J., AND KLINE, C. S. The pdp-11 virtual machine architecture: A case study. In *SOSP '75: Proceedings of the fifth ACM symposium on operating systems principles* (New York, NY, USA, 1975), ACM Press, pp. 97–105.

[21] ROBIN, J., AND IRVINE, C. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium* (2000).

[22] SILBERSCHATZ, A., AND PETERSON, J. L., Eds. *Operating systems concepts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[23] SMITH, J. E., AND NAIR, R. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2005.

[24] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F. C. M., ANDERSON, A. V., BENNETT, S. M., KAGI, A.,

LEUNG, F. H., AND SMITH, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[25] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 195–209.

[26] WITCHEL, E., AND ROSENBLUM, M. Embra: fast and flexible machine simulation. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on measurement and modeling of computer systems* (New York, NY, USA, 1996), ACM Press, pp. 68–79.

Raw data

Times in seconds (smaller is better; BT/VT times obtained via external time source)

	Native	BT	VT
vpr	136	135	139
gcc	67.5	71.0	74.0
mcf	257	244	248
crafty	62.7	64.7	64.9
parser	180	186	187
eon	94	97.2	97.7
perlbmk	115	116	123
gap	71.4	73.7	74.4
vortex	113	116	118
bzip2	114	124	125
twolf	198	210	210
gzip	130	135	139
gcc	67.5	71.0	73.9
gap	73.0	73.7	74.4
kerncomp	265	393	483
kerncomp "Core"	204	259.3	281
forkwait	6.02	36.95	106.4
forkwait "Core"	2.62	18.43	38.89

Passmark ops/sec (larger is better)

	Native	BT	VT
Large RAM	335.44	175.00	67.13
2D Graphics	589.86	373.78	413.05

ApacheBench requests/second (larger is better)
RedHat Enterprise Linux 3

#processes	Native	BT	VT
1	718.24	684.38	584.24
8	1818.85	991.55	840.38
32	1206.01	999.92	815.49
64	2071.09	1014.92	802.05
128	2201.90	985.28	845.12
256	250.94	48.23	52.55

ApacheBench requests/second (larger is better)
WinNetEnterprise

#threads	Native	BT	VT
1	616.48	367.98	430.00
8	815.79	431.10	558.96
32	811.84	431.68	556.53
64	808.93	430.17	525.35
128	803.39	428.70	537.93
256	807.88	426.28	530.16

SPECjbb score (larger is better)
WinNetEnterprise

	Native	BT	VT
SPECjbb	9608	9430	9528

Nano-benchmarks in cycles (smaller is better)

	Native	BT	VT
syscall	242	2308	246
in	3209	216	15826
cr8write	150	35	177
call/ret	11	51	11
pgfault	1093	3927	11242
divzero	889	3223	1014
ptemod: unchanged	1	391	12733
ptemod: P->NP	4	17	9154 (*)
ptemod: NP->P	5	2121	17020 (*)

(*) Unused in main body of paper.