# Software Techniques for Avoiding Hardware Virtualization Exits

Ole Agesen
*VMware*
agesen@vmware.com

Jim Mattson
*VMware*
jmattson@vmware.com

Radu Rugina
*VMware*
rrugina@vmware.com

Jeffrey Sheldon
*VMware*
jeffshel@vmware.com

## Abstract

On modern processors, hardware-assisted virtualization outperforms binary translation for most workloads. But hardware virtualization has a potential problem: virtualization exits are expensive. While hardware virtualization executes guest instructions at native speed, guest/VMM transitions can sap performance. Hardware designers attacked this problem both by reducing guest/VMM transition costs and by adding architectural extensions such as nested paging support to avoid exits.

This paper proposes complementary software techniques for reducing the exit frequency. In the simplest form, our VMM inspects guest code dynamically to detect back-to-back pairs of instructions that both exit. By handling a pair of instructions when the first one exits, we save 50% of the transition costs. Then, we generalize from pairs to *clusters* of instructions that may include loops and other control flow. We use a binary translator to generate, and cache, custom translations for handling exits. The analysis cost is paid once, when the translation is generated, but amortized over all future executions.

Our techniques have been fully implemented and validated in recent versions of VMware products. We show that clusters consistently reduce the number of exits for all examined workloads. When execution is dominated by exit costs, this translates into measurable runtime improvements. Most importantly, clusters enable substantial gains for nested virtual machines, delivering speedups as high as 1.52x. Intuitively, this result stems from the fact that transitions between the inner guest and VMM are extremely costly, as they are implemented in software by the outer VMM.

## 1 Introduction

Early x86 processors were not classically virtualizable because some sensitive instructions were not amenable to trap-and-emulate style execution [14]. On such processors, efficient virtualization required the use of *binary translation* to handle all supervisory level code. The binary translator replaced the sensitive guest instructions with the appropriate code to emulate their behavior in the context of the virtual machine (VM). Most contemporary x86 processors now offer *hardware virtualization* (HV) extensions [3, 10]. These processor extensions introduce a new operating mode, *guest mode*, specifically designed to support VM execution using a trap-and-emulate approach. Binary translation is no longer necessary.

While the processor operates in guest mode, most instructions execute at or near native speed. When the processor encounters an instruction or event of interest to the *virtual machine monitor* (VMM), it *exits* from guest mode back to the VMM. The VMM emulates the instruction or other event, at a fraction of native speed, and then returns to guest mode. The transitions from guest mode to the VMM and back again are high latency operations, during which guest execution is completely stalled.

The first generation of processors with HV support exhibited lackluster performance, largely due to high exit latency and high exit rates [1]. However, from the Pentium 4 (Prescott) to the second generation Intel Core Family processor (Sandy Bridge), hardware advances and microcode improvements have reduced exit latencies by about 80%; see Table 1. Moreover, hardware support for MMU virtualization has reduced the exit rate for many workloads by as much as 90%.

| Microarchitecture | Launch Date | Cycles |
|---|---|---|
| Prescott | 3Q2005 | 3963 |
| Merom | 2Q2006 | 1579 |
| Penryn | 1Q2008 | 1266 |
| Nehalem | 3Q2009 | 1009 |
| Westmere | 1Q2010 | 761 |
| Sandy Bridge | 1Q2011 | 784 |

Table 1: Hardware round-trip latency.

While hardware implementors have gradually improved exit latencies and rates over the past six years, there is still room for further improvement. The main contributions of this paper are a set of software techniques that, primarily, aim to eliminate exits and, secondarily, allow for somewhat faster handling of exits. The software techniques grew out of our previous work on use of binary translation for virtualization and exploit the observation that exits frequently exhibit temporal locality to reduce exit rate. We identify consecutive *pairs* of instructions that would normally cause back-to-back exits and generate a combined translation to circumvent the second exit. More generally, we identify *clusters* of instructions that would normally cause multiple exits and translate them together to avoid all of the exits save the first. As these translations grow more sophisticated, we expose more opportunities to reduce the exit rate.

The rest of this paper is organized as follows. First, Section 2 describes software techniques for optimizing single instruction exit handling. Our approach draws upon previous experience with using binary translation for efficient instruction simulation. The subsequent sections describe improvements by going beyond single instruction exit handling: Section 3 generalizes to two back-to-back instructions, Section 4 further generalizes to multiple statically recognizable instructions, Section 5 adds use of dynamic information, and Section 6 discusses opportunities involving control flow. Section 7 discusses translation reuse safety. Section 8 applies the techniques to nested virtualization. Finally, Section 9 presents an overall performance evaluation, Section 10 presents related work, and Section 11 offers suggestions for promising future work and our concluding remarks.

## 2 Exit Handling Speedup

Recent x86 processors from both Intel and AMD provide hardware support to assist a VMM run a virtual machine. Both implementations allow the VMM to directly execute the guest instruction stream on the CPU using guest mode, which disallows operations that require VMM assistance. If the guest attempts such an operation, an exit occurs in which the CPU suspends guest execution, saves the guest context in a format accessible to the VMM, restores the context of the VMM, and starts executing the VMM. The VMM is then responsible for driving guest execution forward past the operation that caused the exit and then resuming direct execution in guest mode. The challenge for the VMM is to handle such exits quickly.

Handling an exit involves driving guest progress forward past the event that caused the exit. Typically this requires interpreting a single guest instruction. Naturally, this involves knowing what the instruction is. In many cases the only way of determining the instruction from the guest context is to take the guest's instruction pointer (`%rip`), read guest memory (while honoring segmentation and page tables), and decode the instruction bytes. This is a complex and time consuming process. In some cases, which vary from CPU to CPU, hints in the guest context suffice to permit the VMM to step the exiting instruction without directly decoding it. However, even on new CPUs a high percentage of the dynamic exits do not provide enough information.

To address such decoding overhead, the VMM can cache decoded instructions keyed by guest `%rip` (possibly combined with other easy to access guest state). When an exit occurs, we can hash `%rip`, find the predecoded instruction, verify that the guest still contains the same raw instruction bytes as the cache, and proceed to step the instruction using the cached information.

Years of working on virtualization using a binary translator inspired us to take this decoded instruction caching a step further. Rather than just creating a cache of decoded instructions, we generate executable code in a translation cache for each instruction we want to cache. Our hash function then provides us with an address that we simply jump to. The code at that location verifies that the pre-decoded instruction information matches before running the translated code to handle the exit.

This arrangement allows us to reduce the cost of interpretation by using a form of *specialization*. For instance, most interpreters start out with a large switch statement based on instruction opcode. But since the instruction is fixed for any given translation, our translation can directly embed the correct handler. In a similar manner we are able to further specialize the translation with information about the addressing mode, which guest physical address contains the instruction bytes, or even predictions based on the past behavior of the instruction. This approach provides us with a fast exit handler specialized for each frequently exiting guest instruction.

## 3 Handling Exit Pairs

When hardware-assisted page table virtualization is unavailable, exits associated with maintaining shadow page table coherency can be among the most frequent [2]. Typically, 32 bit x86 operating systems use Physical Address Extension (PAE) mode to enable addressing of more than 4 GB of physical memory. In PAE mode, page table entries (PTEs) are 64 bits in size. Most operating systems, including Windows and Linux, update these 64 bit PTEs using two back-to-back 32 bit writes to memory such as the sequence, from a Linux VM, shown below. This results in two costly exits:[1]

```
*   MOV   4(%ecx),%esi ; write top half of PTE
*   MOV   (%ecx),%ebx  ; write bottom half
```

[1] In the examples, exiting instructions are marked with an asterisk.

When we generate the translation for the first exiting instruction we decode the next instruction. If the next instruction can be shown to always access adjacent bytes in memory, we create a *pair* translation which merges both instructions to act as a single 64 bit write to memory. The details of recognizing adjacent writes are straightforward: in the above case, we simply note that the two instructions use the same base register, `%ecx`, with respective displacements (0 and 4) that differ by the width of the memory operand (4 bytes). Other cases, including instructions that combine base and index registers, absolute addresses, and `%rip`-relative addresses follow the same approach.

Combining two instructions into a single block avoids taking the second exit, thereby eliminating half of the hardware overhead. But it also reduces software overheads. Each time a guest writes to a PTE, the VMM must make the corresponding adjustment to the shadow page tables. By treating the pair of guest instructions as a single 64 bit write we can transition the shadow page tables directly to the new state in one step. With a 32 bit Linux guest using PAE, the combined reduction in overheads reduces the time it takes to compile a kernel by 12%.

## 4 Static Cluster Formation

When generating a translation for an exiting instruction, it is straightforward to decode ahead in the guest instruction stream to look for later instructions that will cause subsequent exits. Our translator scans forward a small number of guest instructions, 16 in our implementation, starting from the exiting instruction. It then analyzes the decoded instructions and tries to form a cluster that covers the stretch of guest code from the first exiting instruction to the last one (and no further). If such a cluster is found within the decoded instructions, a translation is emitted for it.

For example, consider this sequence of 16 bit BIOS instructions:

```
* OUT   %eax,%dx
* OUT   $0xed,%al
  MOV   %dx,$0xcfc
  MOV   %al,%cl
  AND   %al,$0x3
  ADD   %dl,%al
  XCHG  %ecx,%eax
  XCHG  %ah,%al
* IN    %al,%dx
  XCHG  %al,%ah
  XOR   %cl,%cl
  JMP   %bx
```

We took an exit on the first OUT instruction. Clearly, it is beneficial to translate (and execute) the second OUT as well. Moreover, by executing through six ALU instructions (which never exit) we can reach an IN. We call such non-exiting instructions *gap fillers* because they fill the gaps between exiting instructions. In this example, the optimal translation covers the first nine instructions but omits the last three for which no benefits are to be had. At runtime, we take an exit on the first OUT and resume after the IN, avoiding two out of three exits to net a 3x execution speedup.

Is this case a rare oddity? No, most guests contain dozens if not hundreds of similar basic blocks. For example, Linux kernels of a certain vintage worked around a chipset timing bug by piggybacking an extra OUT on every IN and OUT. For a native execution, the cost of the extra OUT, while measurable, is affordable. However, in a VM, the work-around is much more expensive because it doubles the number of IN/OUT related exits. Here's an example:

```
* IN    %al,%dx
* OUT   $0x80,%al     ; bug workaround
  MOV   %al,%cl
  MOV   %dl,$0xc0
* OUT   %al,%dx
* OUT   $0x80,%al     ; bug workaround
* OUT   %al,%dx
* OUT   $0x80,%al     ; bug workaround
```

With clustering, not only do we overcome the exit count increase due to the bug work-around, but we also avoid individual exits on the three required IN/OUT instructions.

Our final example comes from Windows XP running PassMark where the guest's context switching code reprograms debug registers (for reasons unknown to us this PassMark process runs under debugger control):

```
* MOVDR %dr2,%ebx
* MOVDR %dr3,%ecx
  MOV   %ebx,0x308(%edi)
  MOV   %ecx,0x30c(%edi)
* MOVDR %dr6,%ebx
* MOVDR %dr7,%ecx
```

In a PassMark 2D run, this cluster and other similar ones, compress 46 million would-be exits down to just 12 million actual exits, improving the benchmark score by 50–80% (depending on the CPU used).

The PassMark example has two memory-accessing gap fillers. When we pull memory accesses out of direct execution and into translated code in the VMM's context, we can no longer use x86 segmentation and paging hardware. Instead, address translation must be done in software, slowing down memory accesses. To prevent gap filler overheads from overwhelming exit avoidance savings, we cap the number of memory accesses between exiting instructions at four.

## 5 Dynamic Cluster Formation

Instructions of types that always exit are *strongly-exiting*. For our VMM, these instruction include CPUID, OUT and HLT. Capturing strongly-exiting instructions in a cluster simply requires decoding forward from the exiting instruction. However, many exits are caused by *weakly-exiting* instructions: loads, stores and read-modify-write instructions that target either memory with traces (such as page tables) or memory-mapped devices. In these cases, inspection of the instruction itself cannot reliably determine whether it will exit or should be treated as a gap filler. Consider this basic block from SUSE Linux Enterprise Server 10, where we have observed an exit on the first instruction:

```
* MOV   -0x201000(%rax),%edx
  MOV   %eax,-0x7fc4215c(8*%rsi)
  AND   %eax,$0xfff
  SUB   %rax,%rcx
* MOV   -0x200ff0(%rax),%r8d
  MOV   %eax,-0x7fc4215c(8*%rsi)
  MOV   %edx,0x60(%rsp)
  AND   %eax,$0xfff
  SUB   %rax,%rcx
* MOV   -0x201000(%rax),%edi
  MOV   %eax,-0x7fc4215c(8*%rsi)
  AND   %eax,$0xfff
  SUB   %rax,%rcx
* MOV   -0x200ff0(%rax),%edx
  MOV   %rax,0x21f1a1(%rip)
```

It may be possible to prove that a downstream instruction *must* exit by starting from the fact that the initial instruction did so. For example, one could use forward data-flow analysis to find relationships between operands of the first instruction and operands of the downstream instructions. Exit pairs, described in Section 3 are a degenerate and successful example of this type of analysis, but the general problem is much harder and is best solved by approximation.

Instead of attempting static analysis of guest code, we have found that a dynamic prediction-based approach is both simpler to implement and more powerful: it can with high accuracy predict instructions that are likely to exit most of the time, and do so without knowing anything about x86 instruction semantics, 16/32/64 bit code, segmentation, etc. In the above example, the instructions marked with an asterisk were predicted to exit so the optimal translation unit *excludes* the last instruction as it will execute faster directly in the context of the guest.

To obtain a dynamic prediction we defer translation until the third time an instruction exits, handling the first two exits with an interpreter and recording untranslated exiting instructions. Then, when we eventually translate, we will have observed two executions of the downstream instructions from the exiting instruction, allowing the use of past behavior as a predictor of the future. This permits us to form clusters that avoid a large number of exits without being too greedy (which would incur overheads from translated execution of non-exiting instructions in a cluster suffix).

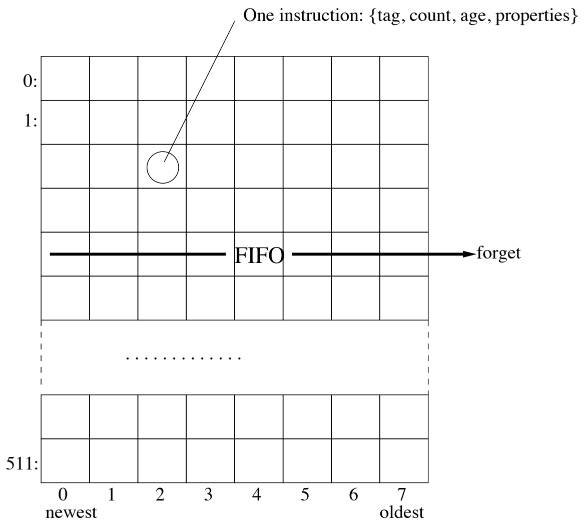


Figure 1: Exit tracking using leakage-rate aware FIFOs.

The exit-tracking data structure must be compact, as is true for any VMM data structure, and offer efficient guest instruction lookup since it is on the critical path for exit handling prior to translation. A fixed-size hash table can meet these requirements, but it leaves one problem unaddressed: instructions may leak due to finite capacity, and if that happens, how do we ensure that we will eventually get the guest exit working set translated? Failure to "count to three exits" (due to hash table leakage) could leave us perpetually interpreting exits, suffering an unacceptable performance loss. For example, a guest reboot loop where the guest uses address space randomization could cause exit-tracking leakage since each reboot incarnation of the guest would look different and impose a new exit working set upon the VMM. In a pathological case, entries could leak from the hash table so fast that some instructions' execution count fail to reach the translation threshold, forcing the VMM to interpret on every exit and, thus, resulting in severe loss of performance.

Instead of a simple hash table, we combine hashing with an array of 8-deep FIFO queues to track exiting instructions. An exiting instruction is hashed into a 35 bit value (using the program counter, length, and instruction bytes). This 35 bit value is split into a 9 bit index and a 26 bit tag. The index selects a row in a 512-long array of FIFOs where each element in the FIFO has a 26 bit tag, a one-bit saturating counter, a 4 bit relative age (time since last exit), and one or more property bits (see below). Figure 1 illustrates this data structure. In response to an exit we hash the instruction and traverse the resulting row in the array of FIFOs to look for a matching tag. The first time an instruction exits, we insert it in the FIFO with a count of zero, shifting out the oldest instruction in the FIFO. The second time an instruction exits, we change its count from zero to one. The third time an instruction exits, we translate it.

To guard against the performance cliff that results if we fail to ever translate (interpreting every exit), when we drop the oldest element from the FIFO, we look at its age. If the age is younger than a set threshold (think of it as 10 seconds on a 2 GHz CPU), we must be "leaking entries fast" and in response we force translation regardless of exit count. The highest rate we may interpret without disabling the threshold requirement is $2*8$ instructions per FIFO per 10 seconds. Multiplied by the number of FIFOs, we have an affordable maximal rate of interpretation: 820 Hz.

The forced translation may cause loss of optimization opportunities, but it avoids falling over the much deeper cliff of interpreting an unbounded number of exits. In practice, we have not encountered any guest that forces us into immediate translation and the resulting loss of optimization opportunities.

We can go one step further than merely using this FIFO data structure to predict which instructions will exit. Since each instruction will be interpreted at least twice before being translated, we can observe not just the fact that instructions exit but *why* they do so. For example, we could record which device they access (if they are device accessing instructions), or whether they access traced memory. In our experience with a large number of common guest operating systems, any given instruction tends to access either one device or another or page tables, but not a variety of such. This means that for most instructions, we can use our FIFO data structure to issue a reliable prediction that not only will the instruction be likely to exit, but also the reason for it to exit. Even if we don't cluster a particular instruction, knowing the likely cause for it to exit allows us to generate better code. Today, we track accesses to the Advanced Programmable Interrupt Controller (APIC), allowing us to speed up guest interrupt processing by a measurable amount. This same idea could be extended to recognizing accesses to other memory-mapped devices that are deemed sufficiently important, such as network interface cards, SCSI devices, etc.

## 6 Control-Flow in Clusters

In many cases, exiting instructions are separated by control-flow instructions. Extending clusters from straight sequences of instructions (i.e., basic blocks) to code sequences with arbitrary control-flow can thus further increase the benefits of clustering. However, including control-flow instructions must be done carefully to avoid costs that outweigh the benefits of clustering.

We describe three increasingly powerful approaches. Our implementation gradually evolved through these different methods over time.

### 6.1 Method 1: No Intra-Cluster Control Flow

The simplest approach allows branch instructions inside clusters but treats all taken branches as cluster termination points. At runtime, a prefix of the cluster will be executed. For instance, the execution of the cluster below would either reach the second exiting `IN` instruction if the `JNZ` branch instruction falls through, or would terminate the cluster in the middle if the branch is taken:

```
* IN    %ax,%dx
  AND   0xd0,$0xfffffeff
  TEST  %ax,$0x20
  JNZ   0x6        --+ M1: terminate if taken
  OR    0xd0,$0x100 |
  MOV   %cx,$0xff <-+ M2: intra-cluster jump
* IN    %ax,%dx
```

### 6.2 Method 2: Forward Branches

A more powerful approach allows intra-cluster forward branches. At runtime, some of the instructions in the cluster may be skipped if forward branches are taken. In the example above, execution reaches the last exiting `IN` instruction regardless of the path taken at the `JNZ` instruction. Forward control-flow provides a simple mechanism to ensure we bound the amount of time spent outside of direct execution.

To generate efficient code, the translator does not update the virtual `%rip` at each instruction inside the cluster. Instead, it updates it at cluster termination and before calling service routines that may cause early termination. As such, the translation of intra-cluster jumps requires additional *glue* code to adjust the delta `%rip` amount from the source instruction to the target.

### 6.3 Method 3: Loops

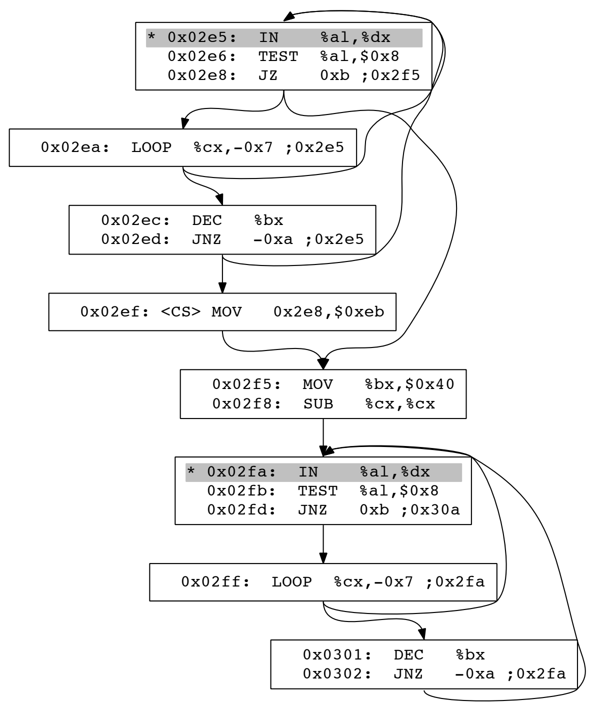
The most general approach is to allow arbitrary branches within a cluster, including control-flow backedges. Although branches to prior instructions in the instruction stream do not necessarily imply loops, in practice all backedges we have encountered correspond to cluster loops. Below is an example of a cluster containing two small loops at the beginning, followed by more exiting instructions:

```
* IN    %al,%dx  <---+
  TEST  %al,$0x8     |
  JNZ   0xfb     ----+
* IN    %al,%dx  <---+
  TEST  %al,$0x8     |
  JNZ   0xfb     ----+
  MOV   %dx,%bx
  MOV   %al,0x18(%esp)
* OUT   %al,%dx
  INC   %dx
  MOV   %al,0x1c(%esp)
* OUT   %al,%dx
  MOV   %al,0x20(%esp)
* OUT   %al,%dx
```

Including loops in clusters has the potential of saving large numbers of exits in a single cluster execution. For instance, during the 64 bit Ubuntu installation a cluster containing loops executes about 20,000 loop iterations in a single runtime instance. However, with this benefit also come two potential dangers: interrupt starvation and performance degradation due to long executions in translated code.

To avoid both of these dangers, we require that each loop iteration checks for pending interrupts and executes at least one exiting instruction. Instead of implementing a full-blown control flow analysis to examine all paths in the cluster, we designed a much simpler, yet effective analysis: for each backedge, we require that the straight list of instructions starting from its target up to the first control-flow instruction encounters both an exiting instruction and an instruction whose translation checks for interrupts. Otherwise the backedge is disabled: the branch instruction is still included in the cluster, but its branch taken path is treated as a cluster termination point. This approach works well in practice: about 84% of all analyzed backedges meet both the interrupt checking and exiting instruction requirements.

But a danger still lurks in the presence of loops that contain merely weakly exiting instructions. Such loops risk degrading rather than improving performance. Consider a loop containing a memory access instruction that we have identified as exiting (using the exit-tracking structure from Section 5). If only a small fraction of the loop iterations touch traced memory, then only a small number of the dynamic instances of that instruction would require exits. Running the entire loop in the cluster may be less efficient because non-exiting memory accesses run faster in direct execution. To contain this risk, we classify cluster backedges as either strong or weak, depending on whether they are guaranteed to reach a strongly exiting instruction or not. We then cap the number of weak backedges traversed per dynamic cluster execution to a small number, 10 in our implementation, while allowing unlimited traversal of strong back edges. This gives us a good balance between achieving most of the benefits and avoiding severe performance degradation in the worst case: if none of the weakly exiting instructions would cause exits, we quickly leave the cluster; if all of the weakly exiting instructions would trigger exits, we only miss 10% of the exit reduction opportunities for that loop.



```
* 0x02e5:  IN     %al,%dx
  0x02e6:  TEST   %al,$0x8
  0x02e8:  JZ     0xb ;0x2f5

  0x02ea:  LOOP   %cx,-0x7 ;0x2e5

  0x02ec:  DEC    %bx
  0x02ed:  JNZ    -0xa ;0x2e5

  0x02ef: <CS> MOV   0x2e8,$0xeb

  0x02f5:  MOV    %bx,$0x40
  0x02f8:  SUB    %cx,%cx

* 0x02fa:  IN     %al,%dx
  0x02fb:  TEST   %al,$0x8
  0x02fd:  JNZ    0xb ;0x30a

  0x02ff:  LOOP   %cx,-0x7 ;0x2fa

  0x0301:  DEC    %bx
  0x0302:  JNZ    -0xa ;0x2fa
```

Figure 2: A cluster with complex control-flow from Windows ME. The two exiting instructions are shadowed.

In the absence of loops and backedges, clusters always end with an exiting instruction since going further would yield no benefits. This is no longer true for loops: the translator extends clusters up to the last backedge, if any. In particular, it is possible to form a special case of clusters having a single exiting instruction at the beginning and ending with a backedge to the first instruction, such as this code fragment from Windows 7:

```
* MOV   (%rcx),%rdx <--+
  ADD   %rcx,$0x8      |
  DEC   %r9            |
  JNZ   -0xc        ---+
```

Due to their small size, clusters usually have simple control-flow. But this is not always the case. Control-flow can become fairly complex within less than 16 consecutive instructions, as shown in Figure 2.

## 7 Checking Code Consistency

Cluster translations are reused every time an exit matches the code address of the cluster. To ensure safety of reuse we must detect cases where the cluster code changes between translation time and use time. This can happen either if a new piece of code gets mapped in at that address, or in the case of self-modifying code.

To detect code changes, each cluster translation begins with a code coherency checking fragment that dynamically checks each byte of the cluster against its corresponding translation-time value. If a mismatch is detected, the existing translation is thrown away and execution falls back to executing just the current exiting instruction. Subsequent exits at the same address will try to form another cluster.

Coherency checks at cluster entry are necessary, but are not sufficient to protect against code modifications. Even if the cluster code matches at the time when an exit is taken, the code may change *during* the execution of the cluster. In other words, the cluster may be self-modifying. We have encountered such clusters in Windows "PatchGuard" code. To address this second issue we have enhanced the translation of memory writes inside the cluster with checks against the pages being accessed. If a cluster tries to write into one of the pages that the cluster belongs to, the cluster execution is terminated. Because clusters contain a small number of adjacent instructions, they span at most two pages. Hence, self-modifying code checks require at most two page checks.

The reader might have noticed that even the cluster in Figure 2 contains self-modifying code. When the `MOV` instruction at address `0x02ef` executes, it changes the opcode of the first branch at address `0x02e8` from `JZ` to `JMP`. Our cluster runtime checks detect this self-modification and immediately terminate the cluster. The next exit at address `0x2e5` will fail its cluster-entry coherency checks, in turn causing cluster re-translation.

## 8 Nested Virtualization

There is growing interest in *nested virtualization*, where an inner VMM runs inside a virtual machine managed by an outer VMM (see Figure 3). Unfortunately, today's hardware does not provide direct assistance for virtualizing HV, so support for virtual HV must be provided by the outer VMM through software emulation. When a hardware exit occurs, control passes to the outer VMM. The outer VMM must then determine whether it should handle the exit itself, or whether it should forward the exit to the inner VMM [13]. To forward an exit to the inner VMM, the outer VMM must emulate the effects of the exit on the state of the inner VM's virtual CPU. This emulation is extremely slow. With VMware Workstation 8 running on a Sandy Bridge CPU, the virtual hardware round-trip latency for a virtualization exit is 9794 cycles, which is over 10 times worse than the underlying physical hardware; compare with Table 1.

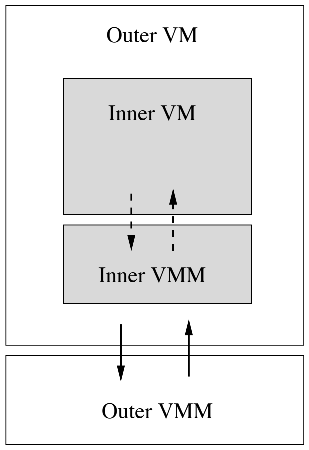


Figure 3: Nested VMs: the inner VM and VMM run inside an outer VM. The solid arrows for the outer VM represent hardware virtualization transitions; the dashed arrows for the inner VM represent emulated hardware virtualization transitions.

Even though the outer VMM gets control of the processor on each hardware exit, we do not try to avoid exits which would have to be forwarded to the inner VMM anyway. When an exit is forwarded to the inner VMM, a large amount of virtual CPU state is modified, including the mappings from linear addresses to physical addresses. Since we do not create translations that span changes in guest paging mode, it is important that the inner VMM exploits all available opportunities for exit avoidance. Every virtual exit that the inner VMM can avoid saves a corresponding hardware exit and the software overhead for forwarding that exit from the outer VMM to the inner VMM.

Exit avoidance in the outer VMM is also significant for nested virtualization. A guest hypervisor often exhibits a higher exit rate than a typical guest, due to frequent modifications of control registers and model specific registers (MSRs), as well as the hardware-assisted virtualization instructions themselves. This is particularly true on Intel hardware. To process an exit, the VMM must access fields in a *virtual machine control structure* (VMCS) that contains the state of the VM. In Intel's implementation, the VMCS is accessed by the new instructions, `VMREAD` and `VMWRITE`, which are strongly-exiting instructions. Typically, a VMM will execute ten or more of these instructions to process a single exit. Thus, a single round-trip from the inner VM to the inner VMM and back again can require ten or more round-trips between the outer VM and the outer VMM, in addition to the emulated exit.

One possible solution to this problem is to paravirtualize VMREAD and VMWRITE [6]. However, clustering in the outer VMM offers some performance benefit without resorting to paravirtualization. For example, in this code sequence from a Hyper-V inner VMM, we save two of three exits:

```
* VMREAD %r9,%rax          ; Exit qualification
  MOV    %eax,$0x2400
* VMREAD %r10,%rax         ; Physical address
  MOV    %eax,$0x4408
* VMREAD %rax,%rax         ; IDT vectoring
```

A clustering-aware inner VMM can help its outer VMM avoid even more exits through dense packing of exiting instructions, *without* creating any incompatibility with unnested execution. This example, from a VMware inner VMM, is executed as part of the exit processing for every (nested) exit:

```
* VMREAD -0x22222a(%rip),%rbx ; RIP
* VMREAD -0x222209(%rip),%rcx ; RSP
* VMREAD %rdx,%rdx            ; RFLAGS
* VMREAD -0x1f229b(%rip),%rbp ; Intr blocking
* VMREAD %rdi,%rdi            ; Exit reason
* VMREAD %rsi,%rsi            ; IDT vectoring
* VMREAD %rbx,%rax            ; CS rights
  ADD    %eax,$0x2
* VMREAD %rbp,%rax            ; SS rights
  TEST   %edi,%edi
  JNZ    0x24
  MOV    %eax,$0x4404
* VMREAD %rax,%rax            ; Interrupt info
```

Each execution of this cluster avoids at least seven hardware exits. By using clustering in the outer VMM and dense packing of exiting instructions in the inner VMM, we have measured a 34% improvement in the time it takes for the inner VMM to process a single strongly-exiting instruction in a nested VM on Intel hardware.

The benefits of clustering in the outer VMM are less significant for nested virtualization on AMD hardware since AMD-V has no equivalent of VMREAD and VMWRITE. However, there are still some opportunities for clustering exits in the outer VMM. Moreover, exit clustering in the inner VMM is just as important on AMD hardware as it is on Intel hardware.

Table 2 illustrates the speedups observed for compiling the Linux kernel in a nested VM with clustering enabled for the inner VMM, the outer VMM, and both VMMs together. Speedups are computed as the ratio of the running time with clusters disabled in both the outer and the inner VM to the running time with clusters. The inner VM runs a 32 bit PAE Linux kernel and the inner VMM uses shadow paging to demonstrate the increased benefits of exit pairs in a nested context. The outer VM runs a 64 bit VMware VMM hosted on a 64 bit Linux OS and the outer VMM uses hardware MMU virtualization.

| | | Inner | | | |
|---|---|---|---|---|---|
| | | Off | | On | |
| | | Intel | AMD | Intel | AMD |
| Outer | Off | — | — | 1.20 | 1.13 |
| | On | 1.27 | 1.06 | 1.52 | 1.19 |

Table 2: Nested kernel compile speedups due to clusters.

## 9 Evaluation

The techniques described in this paper have been fully implemented in VMware products, including ESX, Workstation, and Fusion. The implementation evolved over several years, starting with simpler methods such as just detecting pairs or consecutive IN/OUT instructions in older releases to full support for clusters with arbitrary control-flow and loops in more recent releases. The implementation has therefore been implicitly validated by years of use in the field.

The implementation comprises approximately 10,000 lines of commented C code. This line-count includes the code to translate handlers for single-instruction exits, but excludes functionality that was already present in our VMM for other reasons, such as the x86 instruction decoder, instruction emitters, the translation cache, and fault handlers.

Earlier sections of this paper show performance results for particular techniques along with the description of the technique. These numbers were harvested at the time we implemented the optimization and focused on that one step only. We now step back and look at the aggregate effects of clustering across a number of workloads, including both regular and nested virtual machines.

Figure 4 shows performance data for several variations of Linux kernel compilation (KC), and a nested VM performing Linux kernel compilation (NKC) as well as PassMark 2D graphics. In each case, we run the workload twice, once without clusters and once with clusters. For the KC workloads, we plot the ratio of running time without clusters to running time with clusters. For the NKC cases, as described in Section 8, the nominator is the running time of the nested setup with clustering disabled both in the inner and outer VMM. For PassMark, where a higher score is better, we plot the inverse ratio of scores, i.e. scores with clusters divided by scores without clusters. A ratio greater than 1.0 therefore indicates a performance improvement.

The PassMark test ran in a Windows XP VM using VMware Workstation. The KC benchmarks ran in a SUSE 10.1 VM (either 32 bit using PAE or 64 bit, as indicated). The compiling VM was configured with 2GB of RAM and measurements were taken after a warm-up run so that the workload could run out of the buffer cache. The NKC benchmarks ran in the same SUSE

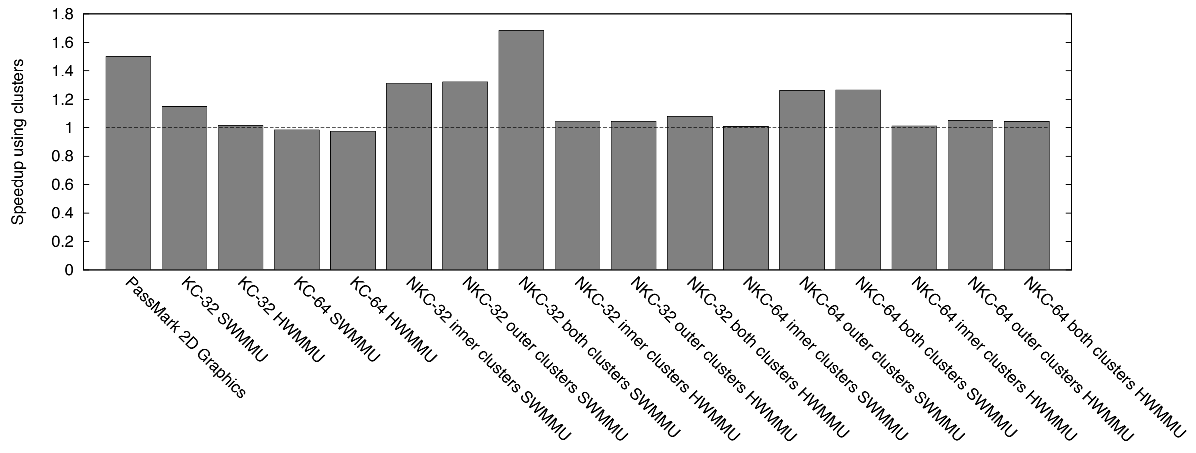


Figure 4: Relative performance improvements using clustering.

10.1 VM (of the indicated code size) running on the same build of VMware Workstation inside an outer 64 bit SUSE 10.1 VM itself running inside VMware Workstation. The outer SUSE 10.1 VM was allocated 4GB of RAM to avoid the need for any disk swapping.

The PassMark result has been previously discussed, and is repeated here for easy comparison, so consider the KC results in the second through fifth column in the figure. We first note that clusters deliver a significant improvement for 32 bit KC when using the software MMU due to dynamically frequent PTE update pairs. The 64 bit KC test, in contrast, shows no gain (the small slowdown is within the noise) as the majority of exits are still due to PTE updates but (1) no PTE pairs exist since a 64 bit VM can (and does) update a PTE with a single 64 bit write and (2) we are unable to cluster from one PTE update to another. While this result may at first blush seem disappointing, the 64 bit KC test runs more than twice as fast as the 32 bit KC test. Put differently, in absolute terms, the 64 bit KC test case has quite good performance from the outset and offers little opportunity for clustering to give it a helping hand. Likewise, when using hardware support for MMU virtualization, clustering provides no benefits whether the guest is 32 bit or 64 bit since this configuration has too low an exit frequency for any exit optimization to matter much.

The rightmost twelve columns in Figure 4 show various nested kernel compile configurations. For NKC workloads we have the option of enabling clustering independently on the inner or the outer VMM. Enabling clustering on just the inner level shows a similar but exaggerated pattern as the non-nested case: 32 bit compiles win significantly while 64 bit compiles show little if any improvement. The larger savings for the nested case result from nested exits being substantially more expensive than non-nested exits.

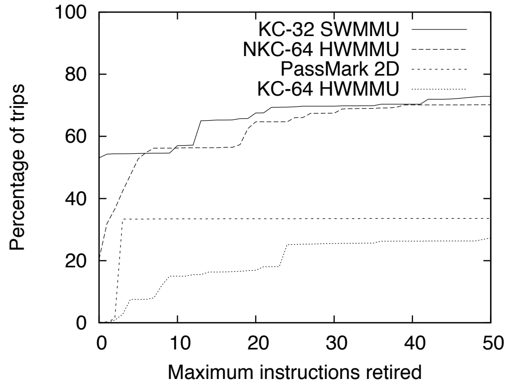


Figure 5: Instructions retired between exits.

Enabling clustering on the outer VMM shows improvements for both 32 bit and 64 bit compiles. Here, instead of avoiding exits from the inner guest, we see benefit from being able to cluster virtualization-related exits made by the inner VMM. This win is visible when using both the software MMU and the hardware MMU, although the win is larger when using the software MMU due to the higher exit rates of the inner guest.

The effects of clustering at the inner and outer levels are not strictly additive but do complement each other as can be best seen by comparing the 32 bit NKC cases.

Clustering fundamentally depends on temporal locality of exits to deliver benefits. With hardware performance counters we can measure the distribution of "distances" between consecutive exits. This data offers an upper bound on the clustering opportunities that exist for an ideal binary translator: one that yields no slowdown over guest mode direct execution and has infinite translation cache capacity. While an ideal translator does not exist, the distribution still offer insights, including suggesting how close to ideal we can reasonably get with translation units capped at 16 instructions.

Figure 5 shows, for a selection of workloads, the cumulative percentage of trips through guest mode where the number of instructions retired is less than or equal to the indicated maximum. PassMark "jumps" to almost 40% at just two instructions, reflecting the extremely high dynamic frequency of the tight debug-register clusters. Even more impressively, 32 bit KC, with the zero-distance PTE pairs, starts out at over 50% of all exits having zero distance, i.e., they must be pairs.

These graphs show that for many workloads a substantial fraction of exits can be captured with simple clusters of at most 16 instructions. However, this does not mean that all workloads benefit meaningfully from clustering: exit rates must be factored in to obtain the "bottom line." For example, the 64 bit KC, which we previously noted shows no measurable runtime benefit from clustering still has 16% of all exits amenable to clustering, but an insufficient exit frequency for it to matter.

Nested VMs, KC, and PassMark all benefit exceptionally well from clustering, and were motivating cases during development. We looked to VMmark 2.0 for testing against other guests, using only the portions of VMmark that involved actually running VMs and omitting meta-tasks like deploying and moving VMs. Since VMmark runs the workloads for fixed duration (including a ramp-up time), we cannot measure speedup with VMmark. Instead, we looked at the reported throughput scores for the individual VMs. We found that results with and without clustering were about the same, factoring in the noise margin. To determine if this null result is due to clustering being ineffective or due to exits being too infrequent for exit avoidance to matter, we measured exit rates per virtual CPU for the VMmark constituent workloads during their steady state period; see Figure 6.

Encouragingly, clustering reduced the exit rates on all of the workloads. But, as the following calculation shows, these workloads are not dominated by exit costs. We assume a combined hardware and software cost of servicing an exit at 3000 cycles. Then, on the 3.33 GHz machine used here, each exit saved by clustering amounts to 0.9 us of CPU time. For the Mail Server workload, clustering saves about 1400 us of CPU time each second, i.e., just 0.14% of a core. So we see a small win, but one that is too small to detect on the bottom line.

While this result may at first seem disappointing, we note three positives: (1) clustering is either a net win or neutral but never a loss; (2) for some latency-sensitive workloads, including request-response client/server workloads and HPC workloads over a low-latency network, even a microsecond may matter; (3) we have found many actual workloads for which clustering delivers a bottom-line visible throughput improvement.

As a final dimension to the performance and effectiveness of clusters, let us relate them to work that has been done in the space of device virtualization. We ran netperf in a VM on VMware Workstation to determine if clusters can improve virtualized network performance. Netperf can measure either throughput or latency. We used the latter, configuring netperf to measure number of network packet roundtrips achievable per second. Meanwhile, in the hypervisor, we counted number of exits per second, allowing us to "normalize" exit counts per network roundtrip. With this setup, we first gave the VM a virtual e1000 NIC. This NIC is an emulated version of a well-known physical NIC produced by Intel. For e1000, with clusters disabled, each network roundtrip induces 2.6 exits. With clustering enabled, the exit count drops by 24% to 1.97 exits per roundtrip.

Replacing e1000 with a paravirtualized vmxnet NIC, a device specifically designed to be virtualization-friendly (i.e., designed to require fewer device touches from guest software) we found that clusters drop the exit rate per roundtrip from 1.3 to 1.2, a reduction of about 8%.

We are encouraged that clusters make a standard e1000 NIC perform measurably better and closer to the level of performance of a paravirtualized NIC. While the latter remains faster, extra steps are needed when switching from an e1000 NIC that works "out of the box" to a paravirtualized NIC that requires an add-on driver. This means that many virtual machines will continue to use an e1000 NIC.

We also find it interesting that a guest running with the paravirtualized NIC benefits nontrivially from clusters. Most likely, the measured improvement can be attributed to guest software outside of the NIC driver per se, perhaps on the general interrupt delivery path, although we have not been able to determine this with certainty.

## 10 Related Work

The cost of an exit has three components: the hardware transition where, typically, microcode switches the CPU from executing guest code to executing hypervisor code, the software transition where hypervisor code saves remaining guest state and loads hypervisor state, and the actual handling of the exiting guest instruction. Moreover, the first two cost components have counter-parts that apply when resuming guest execution. Clusters improve virtual machine performance by avoiding transition costs. They do not significantly speed up the actual handling of instructions that would, absent clustering, have caused their own exits.

Exits can be avoided using hardware or software techniques, or a combination thereof. The history of build-

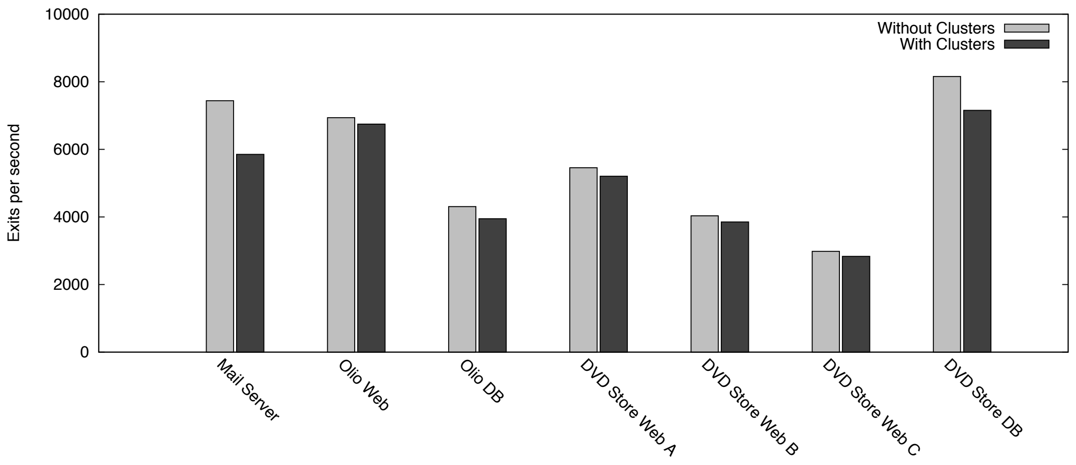


Figure 6: Exit rates for workloads from VMmark.

ing hardware that avoids certain types of exits goes back to the classical IBM mainframe era. Initial mainframe VMMs executed the guest with reduced privileges, causing all privileged instructions to trap. Soon, however, hardware support emerged to avoid many of these traps. For example, IBM's System 370 architecture introduced *interpretive execution* [12], a new hardware execution mode for running guest operating systems. This architectural extension specifies an encoding of guest privileged state and provides a new `SIE` "start interpretive execution" instruction to enable the VMM to hand off guest execution to the actual hardware. Many instructions that would otherwise have trapped in a simple deprivileged execution environment now can run directly in hardware against the provided guest privileged state.

Today, Intel and AMD use the same approach to avoid many virtualization exits. As previously explained, contemporary x86 virtual machines do not merely run in a less privileged protection ring, but run in guest mode with a VMCS (Intel terminology) or a VMCB (AMD terminology) capturing select vCPU state. Intel's `VMRESUME` and AMD's `VMRUN` instructions are equivalent to IBM's `SIE` instruction. As was the case on mainframes, this arrangement allows privileged x86 instructions like `CLI` (disable interrupts) and `STI` (enable interrupts) to execute without taking exits in guest mode: the instructions simply clear and set the vCPU's interrupt flag (`IF`) and check for a pending interrupt whenever `IF` changes from 0 to 1.

Along the same lines, and even more recently, both AMD and Intel have added hardware support for virtualizing the x86 MMU. This hardware support eliminates the need for the VMM to implement shadow page tables to virtualize guest virtual memory. Instead, the RVI/EPT hardware extensions allow the guest and hypervisor, each, to designate a set of page tables that map from guest virtual to guest physical memory and from guest physical memory to host physical memory, respectively. Giving the hardware MMU access to these two sets of page tables significantly reduces the number of exits required for running workloads that have frequent context switches, frequent page faults, and/or frequent memory map/unmap operations [7, 1]. However, these techniques come at the price of higher TLB miss costs and additional MMU implementation complexity.

Turning now to software techniques, paravirtualization [5] was initially used to overcome obstacles to virtualizing the x86 architecture without the need for deeper software techniques such as binary translation [1]. Specifically, if a paravirtualized guest kernel avoids all use of non-trapping privileged instructions such as `POPF`, it can run on a trap-and-emulate VMM. Additionally, paravirtualization was used to improve performance using a batched hypercall interface to amortize the guest/VMM crossing overhead. For example, a bulk update of shadow PTEs was deemed beneficial in the days prior to RVI/EPT hardware support for memory virtualization [4]. Batching of hypercalls resembles our clustering technique, but with the key difference that clusters are "native hardware compatible" whereas (batched) hypercalls only work on a hypervisor and possibly not even across different hypervisors.

Paravirtualization has also been used to accelerate virtual machine I/O. For example, instead of giving the guest operating system a virtual device that behaves exactly like an existing physical device, the VMM may implement a special-purpose NIC that has been developed specifically for the virtualized environment. Such a paravirtualized NIC side-steps the complexity/performance trade-off that a "real" NIC implemented in silicon must obey. It can present an interface to the guest that allows packets to be sent using fewer device "touches" than would be the case for common physical NICs. Xen's device architecture, for example, uses a shared memory area to allow the guest's network driver to communicate asynchronously with the hypervisor [5]. VMware's SVGA graphics device also uses a shared memory FIFO to transmit graphics commands from guest to hypervisor in a low-overhead manner [8].

While device paravirtualization delivers performance gains, it requires that the hypervisor vendor provides not just a VMM but also guest drivers for the paravirtualized devices. Thus, device paravirtualization realizes performance benefits only in exchange for taking on the burden of writing both device emulation code and guest drivers, and probably for multiple guest operating systems. As shown in Section 9, clustering can deliver some of the same benefits as device paravirtualization by reducing the number of exits required to complete an I/O transaction against a standard device.

Pass-through of physical devices to virtual machines eliminates exits in a different manner. With pass-through (also sometimes called "direct device assignment"), guests contain drivers for physical devices, permitting exit-free programming of I/O requests. Gordon et al. [9] describe how a hypervisor can carefully manage physical resources, such as the Interrupt Descriptor Table and the APIC, to further eliminate most interrupt-related exits.

In the Turtles project, Ben-Yehuda et al. took the lead in investigating performance of nested virtualization on Intel x86 processors with VT-x [6]. They discuss virtualization of EPT using EPT shadowing, and discuss device performance extensively, including the pass-through scenario. Impressively, Turtles achieves nested virtual performance within 6–8% of non-nested virtual performance for nontrivial workloads. Much as in our own nested virtualization work, Turtles uses VMCS shadowing to multiplex ("flatten") the inner hypervisor's use of VT-x/EPT onto the underlying physical CPU.

However, as explained earlier, Intel's use of special VMCS accessor instructions, `VMREAD` and `VMWRITE`, presents a particular challenge to nested performance since use of these instructions in the inner hypervisor triggers exits. It appears that to get the best performance, the Turtles project paravirtualizes `VMREAD` and `VMWRITE`, allowing the inner hypervisor to instead use simple memory reads and writes. (This creates exit-behavior similar to what would have been the case on an AMD CPU where VMCB fields are memory mapped.) Since use of paravirtualized VMCS accessors prevents the inner hypervisor from working in a unnested setup, Ben-Yehuda et al. propose use of binary rewriting to convert inner hypervisor `VMREAD` and `VMWRITE` instructions into memory-accesses on the fly. Such code rewriting resembles the dynamic code rewriting techniques proposed by LeVasseur et al. in that they are guest visible [11]. In contrast, our clustering technology does not leave any visible traces that the inner hypervisor can detect, other than indirect timing effects. Moreover, clusters apply not just at the outer hypervisor level (when the inner hypervisor cooperates) but also allows the inner hypervisor to collapse multiple inner guest exits into faster-executing clusters.

## 11 Future Work and Conclusions

The exit avoidance techniques described in this paper have been incorporated in recent VMware ESX, Workstation, and Fusion products. They have been enhanced from release to release and have been running successfully on customer systems for a few years. While we feel that the work has probably reached a sweet spot for today's CPUs and workloads, several directions of future work exist that may take this system to the next level.

First, an adaptive cost/benefit model can more accurately estimate when translations are justified. This would include modeling the costs of expensive translations (e.g., gap-filler memory accesses) and the savings of avoided exits. To capture differences across CPUs or between native and nested executions, the calibration must be done dynamically, e.g., during VMM power-on.

Second, cluster formation can be generalized. While we currently support intra-cluster control-flow, translations are still fairly restrictive: they are small, consecutive, control-flow permits only conditional jumps, and clusters always begin with an exiting instruction. Removing these restrictions permits more powerful clusters such as: loop clusters where the initial exiting instruction is in the middle of the loop; clusters with non-adjacent instructions; and clusters that span call/return control instructions. A branch prediction mechanism could point the translator towards the more frequent execution paths.

Third, caching page walks can reduce the cost of gap-filler memory accesses. For each data memory reference in a cluster, we currently generate a separate walk of the guest page tables to translate the guest linear address to a guest physical address. Often, multiple references to the same guest linear page exist in the same cluster. A simple cache of page translations maintained for the duration of a cluster invocation could reduce the cost of page table walks, in turn permitting us to relax the constraint on the number of memory references in a cluster.

The hardware costs of a virtualization exit have generally improved from one CPU generation to the next. However, the rate of improvement has been slowing. As the hardware round-trip latency bottoms out, efforts to improve virtualization performance must shift to techniques for avoiding virtualization exits. For nested virtualization, exit avoidance is even more important, because virtual hardware is way behind on the curve. Virtualization exits can either be avoided through more sophisticated hardware, or by software techniques such as those described in this paper.

**Acknowledgments.** We would like to thank all current and past members of the VMM team at VMware that contributed to this work with code, discussions, or code reviews. Joshua Schnee and Bruce Herndon helped us with running VMmark. Finally, we thank the reviewers and our colleagues, especially E. Christopher Lewis, for insightful comments on drafts of this paper.

## References

[1] Adams, K., and Agesen, O. 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* (2006), pp. 2–13.

[2] Agesen, O., Garthwaite, A., Sheldon, J., and Subrahmanyam, P. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev. 44*, 4 (Dec. 2010), 3–18.

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

[4] Amsden, Z., Arai, D., Hecht, D., Holler, A., and Subrahmanyam, P. VMI: An interface for paravirtualization. *Ottawa Linux Symposium* (2006).

[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* (2003), pp. 164–177.

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

[7] Bhargava, R., Serebrin, B., Spadini, F., and Manne, S. 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* (2008), pp. 26–35.

[8] Dowty, M., and Sugerman, J. GPU virtualization on VMware's hosted i/o architecture. *SIGOPS Oper. Syst. Rev. 43*, 3 (2009), 73–82.

[9] Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Schuster, A., and Tsafrir, D. Eli: bare-metal performance for i/o virtualization. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS '12, ACM, pp. 411–422.

[10] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, January 2011.

[11] 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.

[12] 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.

[13] Poon, W.-C., and Mok, A. K. 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.

[14] Robin, J. S., and Irvine, C. E. 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.