Heed the Rise of the Virtual Machines

Ole Agesen
VMware

PLDI, June 2012
Innovation in CPU and memory....

Source: Gartner Dataquest, Forecast: DRAM Market Statistics (1Q11)
... Leads to More Processing Power per Server...

Threads per socket (2 per core)  Maximum memory in GB

Source: Forrester Research, *The x86 Server Grows Up And Out* (October 8, 2010)
Virtualization: Run Multiple Workloads on Each Server

- Popek and Goldberg (1974) define VMM:
  - Fidelity
  - Performance
  - Safety/isolation

- Another definition
  - VMM is a layer of indirection
  - Separates physical and virtual hardware
Apply to Entire Data Center

- Aggregate all resources, manage as “Giant Mainframe”
  - Capital efficiency
  - Operational efficiency
  - Flexibility, agility, security, backup, availability, fault tolerance, ...
The World is Now Majority Virtualized

% Virtualized
(of all installed server workloads world wide)

Source: IDC Worldwide Virtualization Tracker, 2010
Not Just Servers

Personal

Corporate
Farming, too
Goal of This Talk

- **Understanding software behavior in a complex stack**
  - Starts with understanding of each layer
  - Then interaction between layers
    - Performance
    - Resource usage
    - Correctness under timing changes
  - All in all: complicated

- **Today: look inside VMM**
  - Explain measurements like:

<table>
<thead>
<tr>
<th>Function</th>
<th>native</th>
<th>virtual</th>
</tr>
</thead>
<tbody>
<tr>
<td>countPrimes</td>
<td>15.0</td>
<td>15.6</td>
</tr>
<tr>
<td>getppid</td>
<td>6.5</td>
<td>29.9</td>
</tr>
</tbody>
</table>
Outline

- Virtualization primer
- Instruction set virtualization
  - Software: binary translation ($BT$)
  - Hardware: Intel VT-x, AMD-V ($HV$)
- Memory virtualization
  - Software: shadow page tables
  - Hardware: AMD RVI, Intel EPT (NPT)
- And beyond . . .
- Conclusions
The Virtual Machine Monitor (VMM) in Context

Resource Management
- CPU Scheduling
- Memory Scheduling
- Storage Bandwidth
- Network Bandwidth

User Worlds support

Mgmt Agent(s) VMX ... VMX

VM

VMM

Distributed Virtual Machine File System

Virtual NIC and Switch

Storage Stack

Network Stack

Device Drivers

ESXi Server

VMkernel Hardware Interface

Hardware

Rest of this talk
Protection Rings (CPL = Current Privilege Level)
Classical “Trap-and-Emulate” Virtualization

• Nonvirtualized (“native”) system
  • OS runs in privileged mode
  • OS “owns” the hardware
  • Application code has less privilege

• Virtualized
  • VMM most privileged (for isolation)
  • Classical “ring compression” or “de-privileging”
    • Run guest OS kernel in Ring 1
    • Privileged instructions trap; emulated by VMM
  • But: does not work for x86 (lack of traps)
Classical VM Performance

• Native speed except for traps
  • Overhead = trap frequency * (average trap cost + average handling cost)

• Trap sources
  • Privileged instructions
  • Page table updates (to support memory virtualization)
  • Memory-mapped devices

• Back-of-the-envelope calculations
  • Trap cost is high on deeply pipelined CPUs: ~1000 cycles
  • Trap frequency is high for “tough” workloads: 100 kHz or greater
  • Bottom line: substantial overhead
Binary Translation of Guest Code

- **No need for traps**
  - Translate away non-virtualizable instructions
  - Replace with VMM runtime calls

- **Popek and Goldberg say “thumbs up!”**
  - Fidelity: instruction-level semantic precision
  - Isolation: translate from full x86 to safe subset
  - Performance: most instructions need no change

- **BT is well-known technology**
  - Smalltalk, Self, JVMs, Shade, Pin, Embra, Dynamo, Transmeta, etc.
BT Mechanics

- Each translator invocation
  - Consume one input Translation Unit
  - Produce one output Compiled Code Fragment

- Store output in Translation Cache
  - Future reuse
  - Amortize translation costs
  - Guest-transparent: no patching “in place”
Translation Unit

- **TU is standard basic block (BB)**
  - Typically ends in control flow
  - Capped at 12 instructions (other restrictions too)
  - Average length 5 instructions

```
4c 8d 68 06 4c 89 ac
24 80 00 00 00 48 89
7c 24 28 48 8b 44 24
78 48 89 44 24 20 4d
8b ce 4c 8b 44 24 40
49 8b d4 48 8d 8c 24
f0 00 00 00 e8 10 1d
f7 ff ...

Decode
```

```
LEA  %r13,0x6(%rax)
MOV  0x80(%rsp),%r13
MOV  0x28(%rsp),%rdi
MOV  %rax,0x78(%rsp)
MOV  0x20(%rsp),%rax
MOV  %r9,%r14
MOV  %r8,0x40(%rsp)
MOV  %rdx,%r12
LEA  %rcx,0xf0(%rsp)
CALL  -0x8e2f0
```
Translation: IDENT

- Most instructions require no change
- Translate by “memcpy”
- No slowdown

<table>
<thead>
<tr>
<th>TU</th>
<th>CCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA %r13, 0x6(%rax)</td>
<td>LEA %r13, 0x6(%rax)</td>
</tr>
<tr>
<td>MOV 0x80(%rsp), %r13</td>
<td>MOV 0x80(%rsp), %r13</td>
</tr>
<tr>
<td>MOV 0x28(%rsp), %rdi</td>
<td>MOV 0x28(%rsp), %rdi</td>
</tr>
<tr>
<td>MOV %rax, 0x78(%rsp)</td>
<td>MOV %rax, 0x78(%rsp)</td>
</tr>
<tr>
<td>MOV 0x20(%rsp), %rax</td>
<td>MOV 0x20(%rsp), %rax</td>
</tr>
<tr>
<td>MOV %r9, %r14</td>
<td>MOV %r9, %r14</td>
</tr>
<tr>
<td>MOV %r8, 0x40(%rsp)</td>
<td>MOV %r8, 0x40(%rsp)</td>
</tr>
<tr>
<td>MOV %rdx, %r12</td>
<td>MOV %rdx, %r12</td>
</tr>
<tr>
<td>LEA %rcx, 0xf0(%rsp)</td>
<td>LEA %rcx, 0xf0(%rsp)</td>
</tr>
<tr>
<td>CALL -0x8e2f0</td>
<td>CALL Translation</td>
</tr>
</tbody>
</table>

0xfffffffffffffff681368
0xfffffffffffffff68136c
0xfffffffffffffff681374
0xfffffffffffffff681379
0xfffffffffffffff68137e
0xfffffffffffffff681383
0xfffffffffffffff681386
0xfffffffffffffff68138b
0xfffffffffffffff68138e
0xfffffffffffffff681396
Translation: CALL

- Push return address on stack

<table>
<thead>
<tr>
<th>TU</th>
<th>CCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA %r13, 0x6(%rax)</td>
<td>LEA %r13, 0x6(%rax)</td>
</tr>
<tr>
<td>MOV 0x80(%rsp),%r13</td>
<td>MOV 0x80(%rsp),%r13</td>
</tr>
<tr>
<td>MOV 0x28(%rsp),%rdi</td>
<td>MOV 0x28(%rsp),%rdi</td>
</tr>
<tr>
<td>MOV %rax, 0x78(%rsp)</td>
<td>MOV %rax, 0x78(%rsp)</td>
</tr>
<tr>
<td>MOV 0x20(%rsp),%rax</td>
<td>MOV 0x20(%rsp),%rax</td>
</tr>
<tr>
<td>MOV %r9,%r14</td>
<td>MOV %r9,%r14</td>
</tr>
<tr>
<td>MOV %r8, 0x40(%rsp)</td>
<td>MOV %r8, 0x40(%rsp)</td>
</tr>
<tr>
<td>MOV %rdx,%r12</td>
<td>MOV %rdx,%r12</td>
</tr>
<tr>
<td>LEA %rcx, 0xf0(%rsp)</td>
<td>LEA %rcx, 0xf0(%rsp)</td>
</tr>
<tr>
<td>CALL -0x8e2f0</td>
<td>MOV %r11,$0x7ff7fed360</td>
</tr>
<tr>
<td></td>
<td>PUSH %r11 ; return address</td>
</tr>
<tr>
<td></td>
<td>MOV -0x25c0ead(%rip),$0x...</td>
</tr>
<tr>
<td>0xfffffffffff6813ad</td>
<td>DISPATCH offs=0xffff71d15</td>
</tr>
</tbody>
</table>
Translation: CALL

- Push return address on stack
- Set up “hint” for fast return

<table>
<thead>
<tr>
<th>TU</th>
<th>CCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA %r13, 0x6(%rax)</td>
<td>LEA %r13, 0x6(%rax)</td>
</tr>
<tr>
<td>MOV 0x80(%rsp), %r13</td>
<td>MOV 0x80(%rsp), %r13</td>
</tr>
<tr>
<td>MOV 0x28(%rsp), %rdi</td>
<td>MOV 0x28(%rsp), %rdi</td>
</tr>
<tr>
<td>MOV %rax, 0x78(%rsp)</td>
<td>MOV %rax, 0x78(%rsp)</td>
</tr>
<tr>
<td>MOV 0x20(%rsp), %rax</td>
<td>MOV 0x20(%rsp), %rax</td>
</tr>
<tr>
<td>MOV %r9, %r14</td>
<td>MOV %r9, %r14</td>
</tr>
<tr>
<td>MOV %r8, 0x40(%rsp)</td>
<td>MOV %r8, 0x40(%rsp)</td>
</tr>
<tr>
<td>MOV %rdx, %r12</td>
<td>MOV %rdx, %r12</td>
</tr>
<tr>
<td>LEA %rcx, 0xf0(%rsp)</td>
<td>LEA %rcx, 0xf0(%rsp)</td>
</tr>
<tr>
<td>CALL -0x8e2f0</td>
<td>&lt;GS&gt;MOV -0x25c0ead(%rip), $0x...</td>
</tr>
<tr>
<td></td>
<td>INCLUDE: MOV $0x7ff7fed8360</td>
</tr>
<tr>
<td></td>
<td>PUSH %r11; return address</td>
</tr>
<tr>
<td></td>
<td>DISPATCH offs=0xffff71d15</td>
</tr>
</tbody>
</table>

Push return address on stack
Set up “hint” for fast return
Translation: CALL

- Push return address on stack
- Set up “hint” for fast return
- Callout to translator to obtain callee

**TU**

<table>
<thead>
<tr>
<th>LEA</th>
<th>%r13,0x6(%rax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>0x80(%rsp),%r13</td>
</tr>
<tr>
<td>MOV</td>
<td>0x28(%rsp),%rdi</td>
</tr>
<tr>
<td>MOV</td>
<td>%rax,0x78(%rsp)</td>
</tr>
<tr>
<td>MOV</td>
<td>0x20(%rsp),%rax</td>
</tr>
<tr>
<td>MOV</td>
<td>%r9,%r14</td>
</tr>
<tr>
<td>MOV</td>
<td>%r8,0x40(%rsp)</td>
</tr>
<tr>
<td>MOV</td>
<td>%rdx,%r12</td>
</tr>
<tr>
<td>LEA</td>
<td>%rcx,0xf0(%rsp)</td>
</tr>
<tr>
<td>CALL</td>
<td>-0x8e2f0</td>
</tr>
</tbody>
</table>

**CCF**

<table>
<thead>
<tr>
<th>0xfffffffffe681368</th>
<th>LEA</th>
<th>%r13,0x6(%rax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xfffffffffe68136c</td>
<td>MOV</td>
<td>0x80(%rsp),%r13</td>
</tr>
<tr>
<td>0xfffffffffe681374</td>
<td>MOV</td>
<td>0x28(%rsp),%rdi</td>
</tr>
<tr>
<td>0xfffffffffe681379</td>
<td>MOV</td>
<td>%rax,0x78(%rsp)</td>
</tr>
<tr>
<td>0xfffffffffe68137e</td>
<td>MOV</td>
<td>0x20(%rsp),%rax</td>
</tr>
<tr>
<td>0xfffffffffe681383</td>
<td>MOV</td>
<td>%r9,%r14</td>
</tr>
<tr>
<td>0xfffffffffe681386</td>
<td>MOV</td>
<td>%r8,0x40(%rsp)</td>
</tr>
<tr>
<td>0xfffffffffe68138b</td>
<td>MOV</td>
<td>%rdx,%r12</td>
</tr>
<tr>
<td>0xfffffffffe68138e</td>
<td>LEA</td>
<td>%rcx,0xf0(%rsp)</td>
</tr>
<tr>
<td>0xfffffffffe681396</td>
<td>MOV</td>
<td>%r11,$0x7ff7fed360</td>
</tr>
<tr>
<td>0xfffffffffe6813a0</td>
<td>PUSH</td>
<td>%r11 ; return address</td>
</tr>
<tr>
<td>0xfffffffffe6813a2</td>
<td>&lt;GS&gt;MOV</td>
<td>-0x25c0ead(%rip),$0x...</td>
</tr>
<tr>
<td>0xfffffffffe6813ad</td>
<td>DISPATCH offs=0xffff71d15</td>
<td></td>
</tr>
</tbody>
</table>
Translation: chaining

- Push return address on stack
- Set up “hint” for fast return
- Callout to translator to obtain callee
- Overwrite callout with callee translation (or insert JMP)

<table>
<thead>
<tr>
<th>TU</th>
<th>CCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA %r13, 0x6(%rax)</td>
<td>LEA %r13, 0x6(%rax)</td>
</tr>
<tr>
<td>MOV 0x80(%rsp), %r13</td>
<td>MOV 0x80(%rsp), %r13</td>
</tr>
<tr>
<td>MOV 0x28(%rsp), %rdi</td>
<td>MOV 0x28(%rsp), %rdi</td>
</tr>
<tr>
<td>MOV %rax, 0x78(%rsp)</td>
<td>MOV %rax, 0x78(%rsp)</td>
</tr>
<tr>
<td>MOV 0x20(%rsp), %rax</td>
<td>MOV 0x20(%rsp), %rax</td>
</tr>
<tr>
<td>MOV %r9, %r14</td>
<td>MOV %r9, %r14</td>
</tr>
<tr>
<td>MOV %r8, 0x40(%rsp)</td>
<td>MOV %r8, 0x40(%rsp)</td>
</tr>
<tr>
<td>MOV %rdx, %r12</td>
<td>MOV %rdx, %r12</td>
</tr>
<tr>
<td>LEA %rcx, 0xf0(%rsp)</td>
<td>LEA %rcx, 0xf0(%rsp)</td>
</tr>
<tr>
<td>CALL -0x8e2f0</td>
<td>MOV %r11, $0x7ff7feda360</td>
</tr>
<tr>
<td></td>
<td>PUSH %r11 ; return address</td>
</tr>
<tr>
<td></td>
<td>&lt;GS&gt;MOV -0x25c0ead(%rip), $0x...</td>
</tr>
</tbody>
</table>

CALLEE TRANSLATION...
Non-IDENT Translations

- All control flow (since translator relocates)
- PC-relative addressing
- Privileged instructions (POPF, CLI, MOVCR, IN/OUT)
- Segment operations
- Access to special memory/addresses
- $%r11 – our work register
Translator Properties

- **Binary**
  - Input is x86 “hex” not source

- **Widening**
  - Output is always 64 bit
  - Input can be real mode, v8086 mode, 16/32/64 bit

- **Dynamic**
  - Interleave translation and execution

- **On-demand**
  - Translate only what we are about to execute

- **System-level**
  - Make no assumptions about guest code
  - Full virtual state recovery on fault/trap/interrupt

- **Adaptive**
  - Observe execution, and retranslate to improve performance
Combining Binary Translation and Direct Execution

Direct Exec (user)

VMM

BT (kernel)

Faults, syscalls, interrupts

IRET, SYSRET
Analysis of a BT-based VMM

• **Costs**
  • Running the translator (minor)
  • Path lengthening: output is sometimes longer than input
  • System call overheads: DE/BT transition

• **Benefits**
  • Avoid costly traps
  • Most instructions need no change (IDENT)
  • Adaptation: adjust translation in response to guest behavior
    • Online profile-guided optimization
  • User-mode code runs at full speed (direct execution)
Performance examples for BT

• **OpenSuse 12.1 64 bit, AMD Phenom II N620, 2.8 GHz, Workstation 8**
  - **Native:** run benchmark on host
  - **DE:** run benchmark in VM using Direct Execution
  - **BT:** run benchmark in VM, forcing use of Binary Translation

• **countPrimes: ALU code -- runs at speed**

```c
int isPrime(int n) {
    if (n % 2 == 0) return n == 2;
    for (int i = 3; i * i <= n; i += 2) {
        if (n % i == 0) return FALSE;
    }
    return TRUE;
}

int c = 0;
for (int n = 2; n < 10000000; n++) {
    if (isPrime(n)) c++;
}  printf("#primes below 10 million: %d\n", c);
```

<table>
<thead>
<tr>
<th></th>
<th>native</th>
<th>DE</th>
<th>BT</th>
</tr>
</thead>
<tbody>
<tr>
<td>countPrimes</td>
<td>15.0</td>
<td>15.6</td>
<td>15.7</td>
</tr>
</tbody>
</table>
Performance examples for BT

- **countPrimes:** ALU code runs at speed
- **Fibonacci:** indirect control flow is slower

\[ \text{fib}(46) = 2971215073 \]

```c
int64_t fib(int64_t n) {
    if (n <= 1) return 1;
    return fib(n - 1) + fib(n - 2);
}

printf("fib(46) = %ld\n", fib(46));
```

<table>
<thead>
<tr>
<th></th>
<th>native</th>
<th>DE</th>
<th>BT</th>
</tr>
</thead>
<tbody>
<tr>
<td>countPrimes</td>
<td>15.0</td>
<td>15.6</td>
<td>15.7</td>
</tr>
<tr>
<td>fib(46)</td>
<td>21.3</td>
<td>21.1</td>
<td>54.7</td>
</tr>
</tbody>
</table>

Fib, *if* it were a system call
Performance examples for BT

- **countPrimes**: ALU code runs at speed
- **Fibonacci**: indirect control flow is slower
- **getppid**: system calls are slower (kernel/user transition itself, not kernel work)

```c
defintion
for (i = 0; i < 10000000000; i++) {
    getppid();
}
```

<table>
<thead>
<tr>
<th>Function</th>
<th>native</th>
<th>DE</th>
<th>BT</th>
</tr>
</thead>
<tbody>
<tr>
<td>countPrimes</td>
<td>15.0</td>
<td>15.6</td>
<td>15.7</td>
</tr>
<tr>
<td>fib(46)</td>
<td>21.3</td>
<td>21.1</td>
<td>54.7</td>
</tr>
<tr>
<td>getppid</td>
<td>6.5</td>
<td>29.9</td>
<td></td>
</tr>
</tbody>
</table>

**Microbenchmarks**
- Not directly representative of real workloads ("undiluted evilness")
Using Hardware Support (HV) to Run Guests (Intel VT-x, AMD-V)

- Key feature: root vs. guest CPU mode
  - VMM executes in root mode
  - Guest (OS and apps) execute in guest mode
  - Hardware-defined VMCS/VMCB holds guest state

- VMM and Guest run as “co-routines”
  - VM enter
  - Guest runs
  - A while later: VM exit
  - VMM runs

- Very much like classical trap-and-emulate
  - Guest-invisible deprivileging
  - All necessary traps
How VMM Controls Guest Execution

- **Hardware-defined structure**
  - Intel: VMCS (virtual machine control structure)
  - AMD: VMCB (virtual machine control block)

- **VMCS/VMCB contains**
  - Guest state
  - Control bits that define conditions for exit
    - Exit on IN, OUT, CPUID, ...
    - Exit on write to control register %cr3
    - Exit on page fault, pending interrupt, ...

- VMM uses control bits to “confine” and observe guest
Analysis of HV-based VMM

- VMM only intervenes to handle exits
- Same performance equation as classical trap-and-emulate:
  - overhead = exit frequency * (average exit cost + average handling cost)
- VMCB/VMCS can avoid simple exits but many remain
  - Page table updates
  - Context switches
  - In/out
  - Interrupts
Virtualization Exits are Expensive but Improving

- Hardware round-trip cost in cycles

<table>
<thead>
<tr>
<th>Micro-architecture</th>
<th>Launch date</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prescott</td>
<td>3Q2005</td>
<td>3963</td>
</tr>
<tr>
<td>Merom</td>
<td>2Q2006</td>
<td>1579</td>
</tr>
<tr>
<td>Penryn</td>
<td>1Q2008</td>
<td>1266</td>
</tr>
<tr>
<td>Nehalem</td>
<td>1Q2009</td>
<td>1009</td>
</tr>
<tr>
<td>Westmere</td>
<td>1Q2010</td>
<td>761</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>1Q2011</td>
<td>784</td>
</tr>
</tbody>
</table>
HV Optimization: Avoiding Exits

- **In hardware:**
  - CLI/STI (disable/enable interrupts)
    - Benign when no interrupt is pending
    - Use hardware to virtualize interrupt flag EFLAGS.IF
  - APIC (advanced programmable interrupt controller)
    - Flex-priority: raise/lower task-priority without exiting up to “threshold”
    - Nested paging avoids MMU-related exits (discussed later)

- **In software: exit clusters**
  - Exploit locality of exiting instructions
  - Translate custom exit handlers to amortize analysis cost

```plaintext
XP PassMark 2D:

* MOVDR %dr2,%ebx
* MOVDR %dr3,%ecx
  MOV %ebx,0x308(%edi)
  MOV %ecx,0x30c(%edi)
* MOVDR %dr6,%ebx
* MOVDR %dr7,%ecx
```
Performance Results for AMD-V

• System calls: things are good again

```c
for (i = 0; i < 10000000000; i++) {
    getppid();
}
```

<table>
<thead>
<tr>
<th></th>
<th>native</th>
<th>BT</th>
<th>HV</th>
</tr>
</thead>
<tbody>
<tr>
<td>getppid</td>
<td>6.5</td>
<td>29.9</td>
<td>6.8</td>
</tr>
</tbody>
</table>
Performance Results for AMD-V

- getppid: system calls are good again
- forkwait: no better

```
for (int i = 0; i < 25000; i++) {
    if (fork() == 0) return;
    wait();
}
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Native</th>
<th>BT</th>
<th>HV</th>
</tr>
</thead>
<tbody>
<tr>
<td>getppid</td>
<td>13.8</td>
<td>114</td>
<td>14.9</td>
</tr>
<tr>
<td>forkwait</td>
<td>5.4</td>
<td>22.8</td>
<td>24.3</td>
</tr>
</tbody>
</table>

- New page tables for new processes
- Copy on write
- Page faults
Memory Virtualization

- Shadow page tables
- Hardware support (AMD RVI, Intel EPT)
Virtual Memory

- Applications see contiguous virtual address space, not physical memory
- OS defines VA→PA mapping
  - Usually at 4 KB granularity
  - Mappings are stored in page tables
- HW memory management unit (MMU)
  - Page table walker
  - TLB (translation look-aside buffer)
Virtualizing Virtual Memory

• To run multiple VMs, another level of memory indirection is needed
  • Guest OS controls virtual to physical mapping: VA→PA
    • No access to machine memory (to enforce isolation)
  • VMM maps guest physical to machine memory: PA→MA

• How to make this go fast?
Software Solution: Shadow Page Tables

- VMM builds “shadow page tables” to accelerate the mappings
  - Composition of VA→PA and PA→MA
  - Shadow directly maps VA→MA
Hardware MMU's Sees Composite Mapping

- Full-speed TLB hit performance
  - TLB caches VA→MA mapping
- Full-speed TLB miss performance
  - Hardware page walker reloads TLB
3-way Performance Trade-off in Shadow Page Tables

1. Trace costs
   - VMM must intercept Guest writes to primary page tables
   - Propagate change into shadow page table (or invalidate)

2. Page fault costs
   - VMM must intercept page faults
   - Validate shadow (hidden page fault), or forward fault to guest (true fault)

3. Context switch costs
   - VMM must intercept CR3 writes
   - Activate new set of shadow page tables

- Finding good trade-off is crucial for performance
- Increasingly difficult with higher VCPU counts
Hardware Solution: Nested Page Tables (NPT)

TLB

<table>
<thead>
<tr>
<th>VA</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TLB fill hardware

Guest PT ptr

Nested PT ptr

VA→PA mapping

PA→MA mapping

VMM

guest
Analysis of NPT

• Dynamic
  • MMU composes VA→PA and PA→MA mappings on the fly at TLB fill time

• Benefits
  • Significant reduction in exit frequency
    • No trace faults (primary page table modifications as fast as native)
    • Page faults require no exits
    • Context switches require no exits
  • No shadow page table memory overhead
  • Better scalability to high vcpu count

• Costs
  • More expensive TLB misses
  • O(n^2) cost for page table walk (n = depth of the page table tree)
Performance Results for AMD-V with RVI

- **getppid:** system calls are still good
- **forkwait:** a decent improvement with NPT

```c
for (int i = 0; i < 25000; i++) {
    if (fork() == 0) return;
    wait();
}
```

<table>
<thead>
<tr>
<th></th>
<th>native</th>
<th>BT</th>
<th>HV</th>
<th>RVI</th>
</tr>
</thead>
<tbody>
<tr>
<td>getppid</td>
<td>6.5</td>
<td>29.9</td>
<td>6.8</td>
<td>6.8</td>
</tr>
<tr>
<td>forkwait</td>
<td>5.4</td>
<td>22.8</td>
<td>24.3</td>
<td>12.6</td>
</tr>
</tbody>
</table>
Performance Results for AMD-V with RVI

- getppid: system calls are still good
- forkwait: a decent improvement with NPT
- memsweep: TLB miss costs are significant with NPT

```c
#define S (8192 * 4096)
volatile char large[S];

for (unsigned i = 0; i < 20 * S; i++) {
    large[(4096 * i + i) % S] = 1 + large[i % S];
}
```

<table>
<thead>
<tr>
<th></th>
<th>native</th>
<th>BT</th>
<th>HV</th>
<th>RVI</th>
</tr>
</thead>
<tbody>
<tr>
<td>getppid</td>
<td>6.5</td>
<td>29.9</td>
<td>6.8</td>
<td>6.8</td>
</tr>
<tr>
<td>forkwait</td>
<td>5.4</td>
<td>22.8</td>
<td>24.3</td>
<td>12.6</td>
</tr>
<tr>
<td>memsweep</td>
<td>12.0</td>
<td>12.8</td>
<td>12.8</td>
<td>26.5</td>
</tr>
</tbody>
</table>

Using 4 KB pages
TLB Performance

- overhead = frequency of miss * cost of servicing miss
- 4 KB pages are too small for many workloads
  - 1024 entry TLB maps only 4 MB of memory
  - Also true natively but especially in VM (expensive nested page walks)
- Use large pages (2 MB or 1 GB)
  - Benefits
    - Fewer page walks (increased TLB capacity)
    - Faster page walks (fewer levels to traverse)
  - Costs
    - Less opportunity for fine-grained memory management
      - Demand fault-in
      - Swap
      - Page sharing
Large Pages to the Rescue: memsweep Details

- 32 MB array, 20 passes over array, 640M loop iters
- 11 instrs in loop, 2 memory accesses
- Loop instr count: 11 * 20 * 32M = 6.875G (99.7%)
- Total instr count (`perf`): 6.892G (100%)
- With 4 KB pages: 652M dtlb misses (48.6% miss rate!)
  - Native runtime: 12.0s, IPC = 0.22
  - VM w. NPT runtime: 26.5s, IPC = 0.10
- With 2 MB pages: 0.9M dtlb misses (0.07% miss rate)
  - Native runtime: 5.1s, IPC = 0.52
  - VM w. NPT runtime estimate(*):
    - 724x reduction in TLB miss frequency (and they get faster too)
    - (26.5 – 5.1) / 724 = 0.3s residual overhead

(*) Estimating because Workstation does not support allocation of large pages on Linux host.
Virtualizing Performance Counters

• CPUs have hardware-based event counters
  • For performance analysis (and debugging)
  • Tools like vtune and perf gather and visualize data
  • Want such tools to work in VMs

• Must virtualize performance counters
  • Let counters run through exits?
  • Stop counters at exit?
  • Best solution depends on type of events collected
    • E.g., cycles vs. branch counts
Other Topics

- **Nested VMs**
  - Implementing vVT-x on VT-x, and vAMD-V on AMD-V

- **De-featuring VCPUs**
  - Hypervisor: hide newer CPU's instructions to allow vMotion to older CPU
  - Guest: software must respect feature bits (no try-and-catch-fault)

- **Page-sharing**
  - Use memory indirection for transparent page sharing

- **Large pages**
  - Hypervisor: defragment memory at “machine level” invisibly to guests
  - Guest: use large pages where appropriate

- **NUMA**
  - Hypervisor: NUMA-aware CPU and memory scheduler/allocator
  - vNUMA: make guest aware of thread/core/cache/memory topology
    - Challenge: dynamically changing vNUMA topology?
Conclusions

- Virtual machines are everywhere
- Strong resemblance to physical
  - Hardware support for instruction execution (VT-x, AMD-V)
  - Hardware support for memory virtualization (RVI, EPT)
  - Optimize for VT-x/EPT and AMD-V/RVI
- “Thin layer of software” – but not so thin that it should be ignored
  - Virtualization exits are still expensive
    - Clustering
  - Memory effects are “interesting”
    - TLB miss costs
    - Large pages
    - Page sharing
    - NUMA and vNUMA
- Develop in VMs for deployment in VMs
  - Tools, applications
References

In another milestone in the shrinkage of what used to be a refrigerator-sized computer, Xenon has been replaced by a new virtual machine hosted on the Computer Science Department's VMware vSphere cluster. We expect you'll find it to be more stable now.
Timeline

VMware founded

1998  1999  2000  2001  2002  2003  2004  2005  2006  2007  2008  2009  2010  2011...

- Workstation 1.0
- ESX Server 1.0
- Workstation 2.0
- ESX 2.0 (vSMP, 2vcpus)
- ESX 3.0 (4 vcpus)
- ESX 3.5
- ESX 4.0 (8 vcpus)
- ESX 4.1
- ESX 5 (32 vcpus)
- Fusion 1.0

- x86-64 Dual core
- Intel VT-x
- Intel EPT
- AMD-V
- AMD RVI
- AMD RVI
- Intel EPT

- Workstation 5.5 (64 bit guests)
### Windows 2000 Boot/Halt Translation Stats

<table>
<thead>
<tr>
<th>#</th>
<th>units</th>
<th>size</th>
<th>instr</th>
<th>cycles</th>
<th>output</th>
<th>size</th>
<th>cyc/ins</th>
<th>ins/unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>38690</td>
<td>336k</td>
<td>120k</td>
<td>252M</td>
<td>924k</td>
<td>2097</td>
<td>3.11</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>48839</td>
<td>500k</td>
<td>169k</td>
<td>318M</td>
<td>1164k</td>
<td>1871</td>
<td>3.48</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>108k</td>
<td>1187k</td>
<td>392k</td>
<td>754M</td>
<td>2589k</td>
<td>1920</td>
<td>3.61</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>29362</td>
<td>264k</td>
<td>89749</td>
<td>287M</td>
<td>951k</td>
<td>3197</td>
<td>3.06</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>96876</td>
<td>1000k</td>
<td>337k</td>
<td>708M</td>
<td>2418k</td>
<td>2100</td>
<td>3.48</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>58553</td>
<td>577k</td>
<td>193k</td>
<td>403M</td>
<td>1572k</td>
<td>2078</td>
<td>3.31</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>19430</td>
<td>148k</td>
<td>50951</td>
<td>148M</td>
<td>633k</td>
<td>2904</td>
<td>2.62</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>13081</td>
<td>87811</td>
<td>30455</td>
<td>124M</td>
<td>494k</td>
<td>4071</td>
<td>2.33</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>-------------</td>
<td>------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>Total</td>
<td>413k</td>
<td>4101k</td>
<td>1384k</td>
<td>2994M</td>
<td>10748k</td>
<td>2161</td>
<td>3.35</td>
<td></td>
</tr>
</tbody>
</table>

---
Nested Virtualization: Running a Hypervisor in a VM

- **Why?**
  - Guest OS may want to use VT-x or AMD-V (XP mode on Windows 7)
  - Demo full data center on a laptop
  - Lab/QA environments at scale

- **Both Intel and AMD architectures are virtualizable**
  - Map vVT-x to VT-x and vAMD-V to AMD-V

- **Main challenge: virtual exits even more expensive than physical**
  - Exit avoidance is paramount
    - Use NPT
    - Exploit clustering opportunities for both Inner and Outer VM

- **Can often achieve nested performance at 50+%**