Heed the Rise of the Virtual Machines

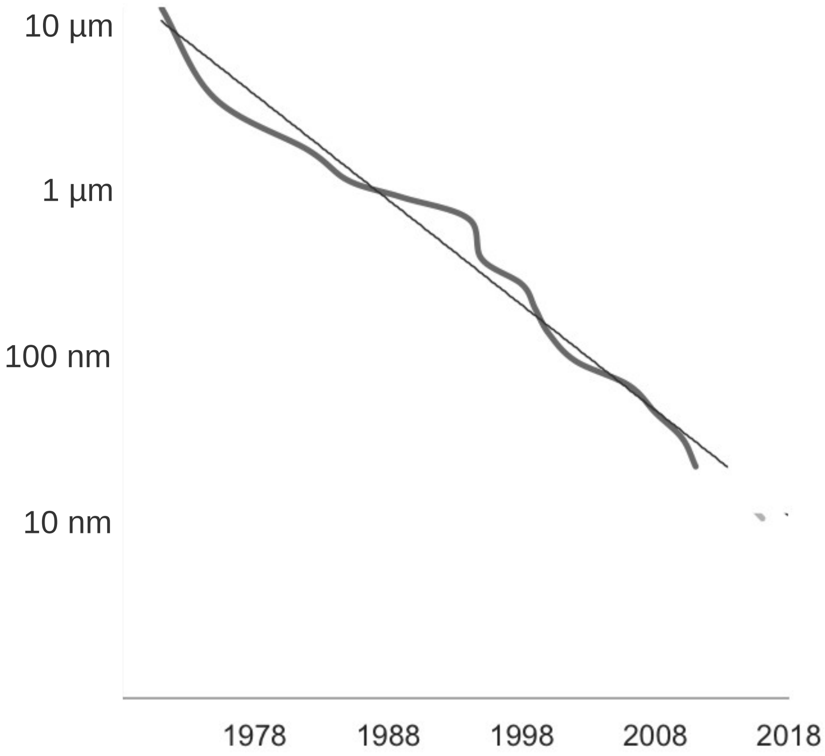
Ole Agesen
VMware

PLDI, June 2012

Innovation in CPU and memory….

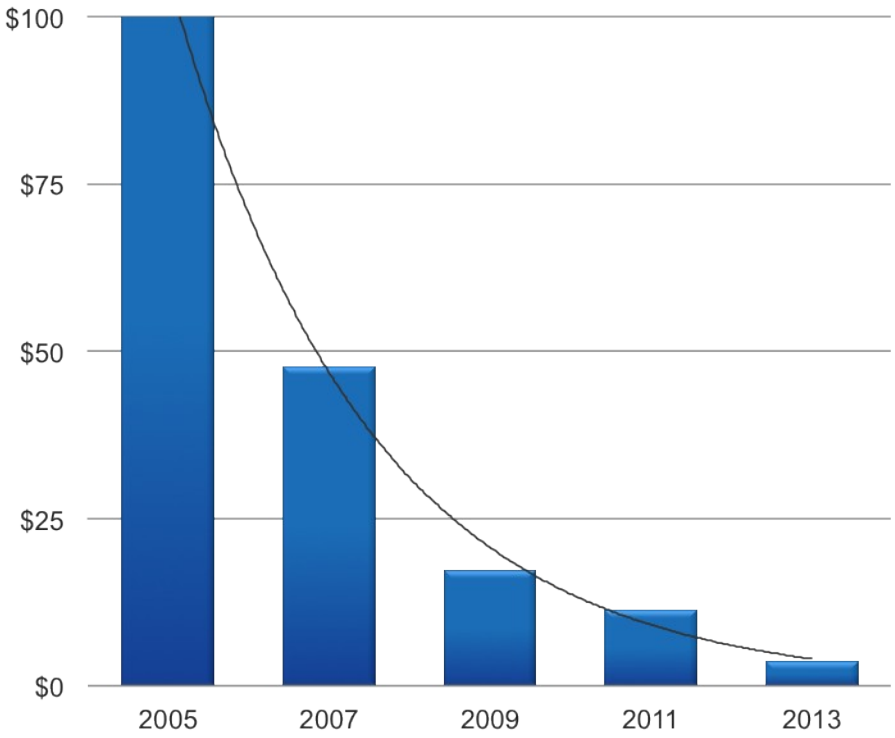
CPU

process technologies



DRAM

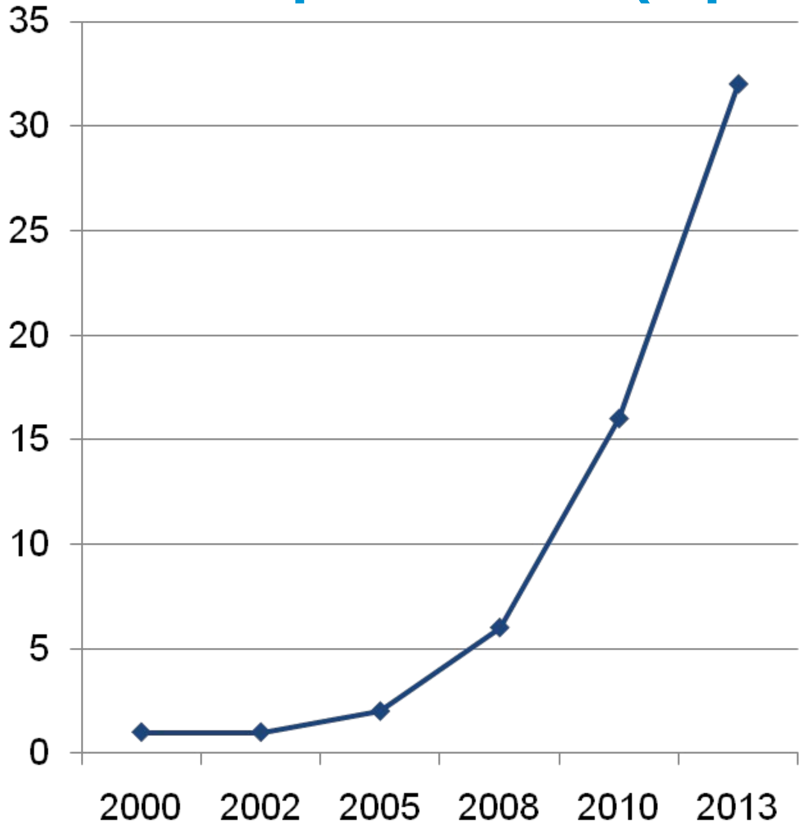
$/GB



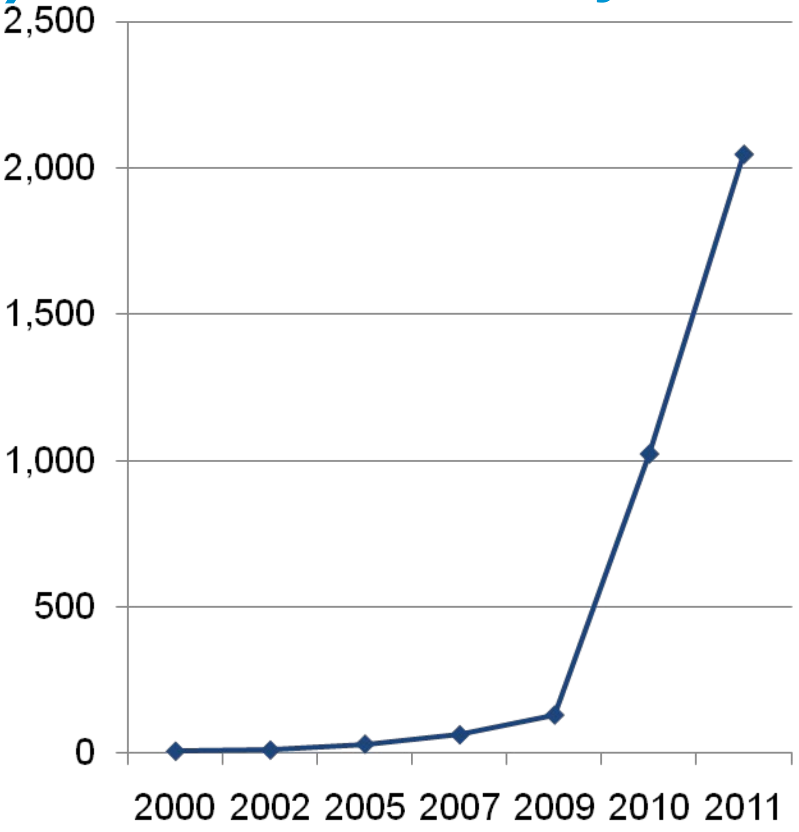
Source: Gartner Dataquest, *Forecast: DRAM Market Statistics* (1Q11)

… Leads to More Processing Power per Server…

Threads per socket (2 per core)

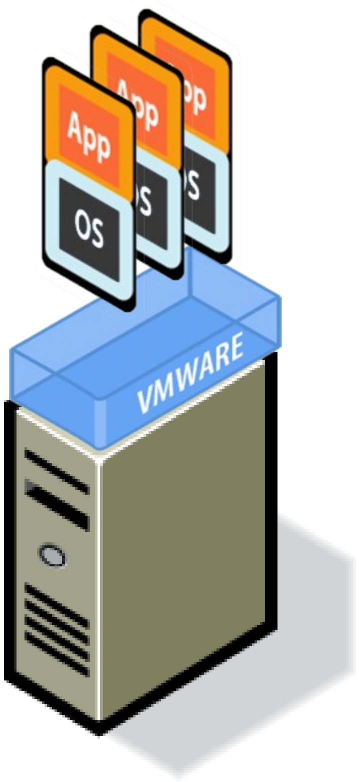


Maximum memory in GB



Source: Forrester Research, *The x86 Server Grows Up And Out* (October 8, 2010)

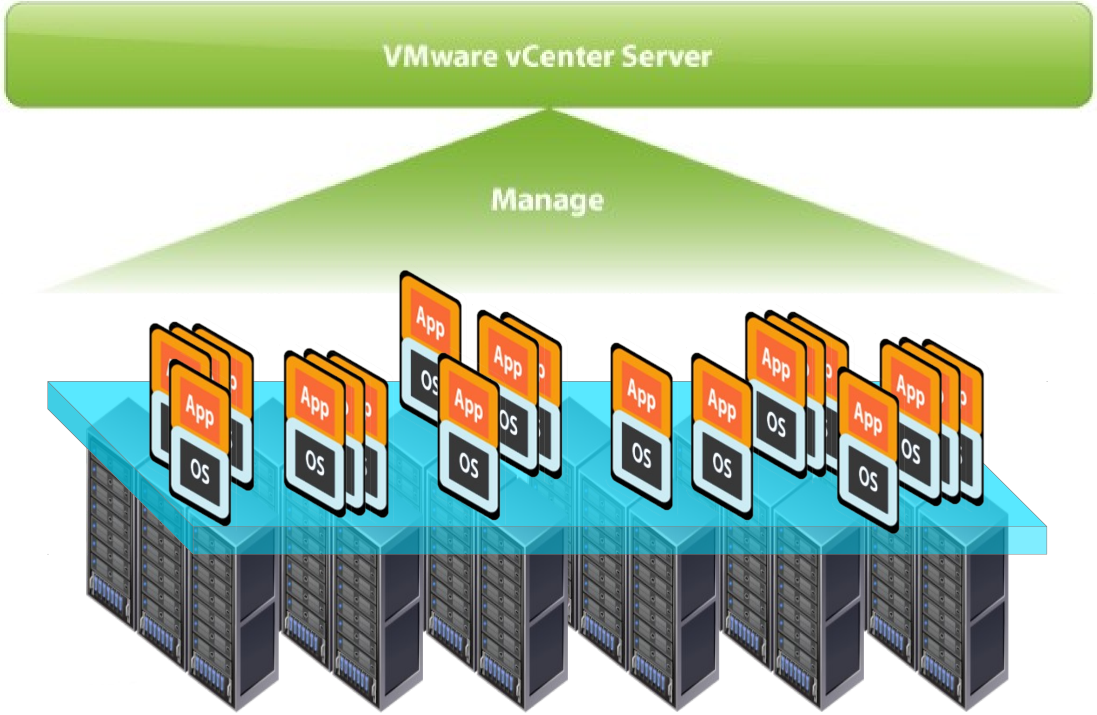
Virtualization: Run Multiple Workloads on Each Server



- **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

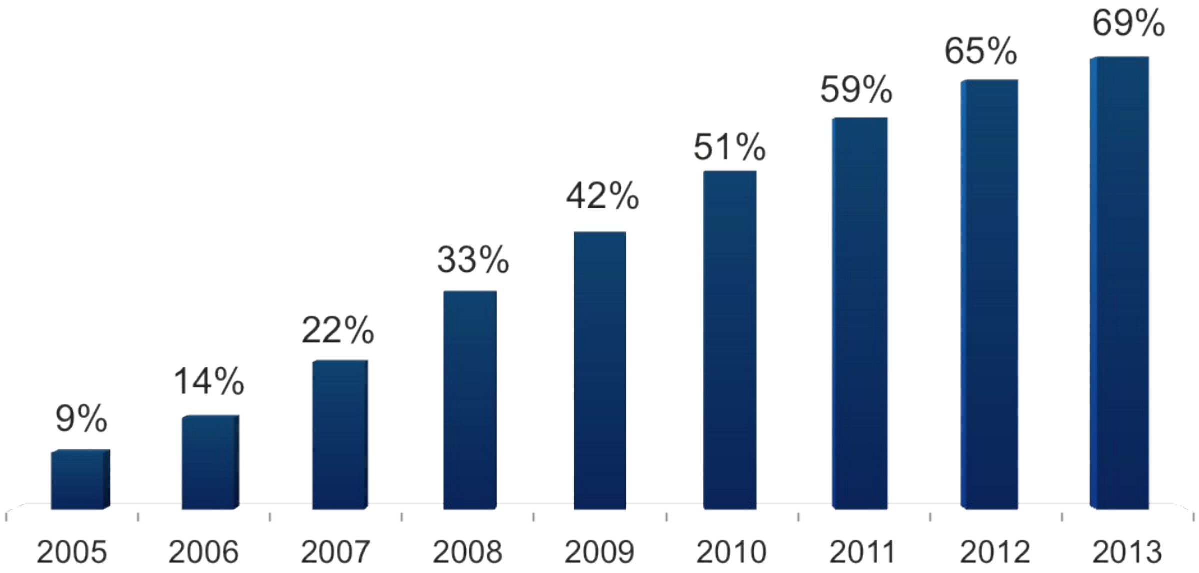
- **Aggregate all resources, manage as "Giant Mainframe"**
  - Capital efficiency
  - Operational efficiency
  - Flexibility, agility, security, backup, availability, fault tolerance, ...



The World is Now Majority Virtualized

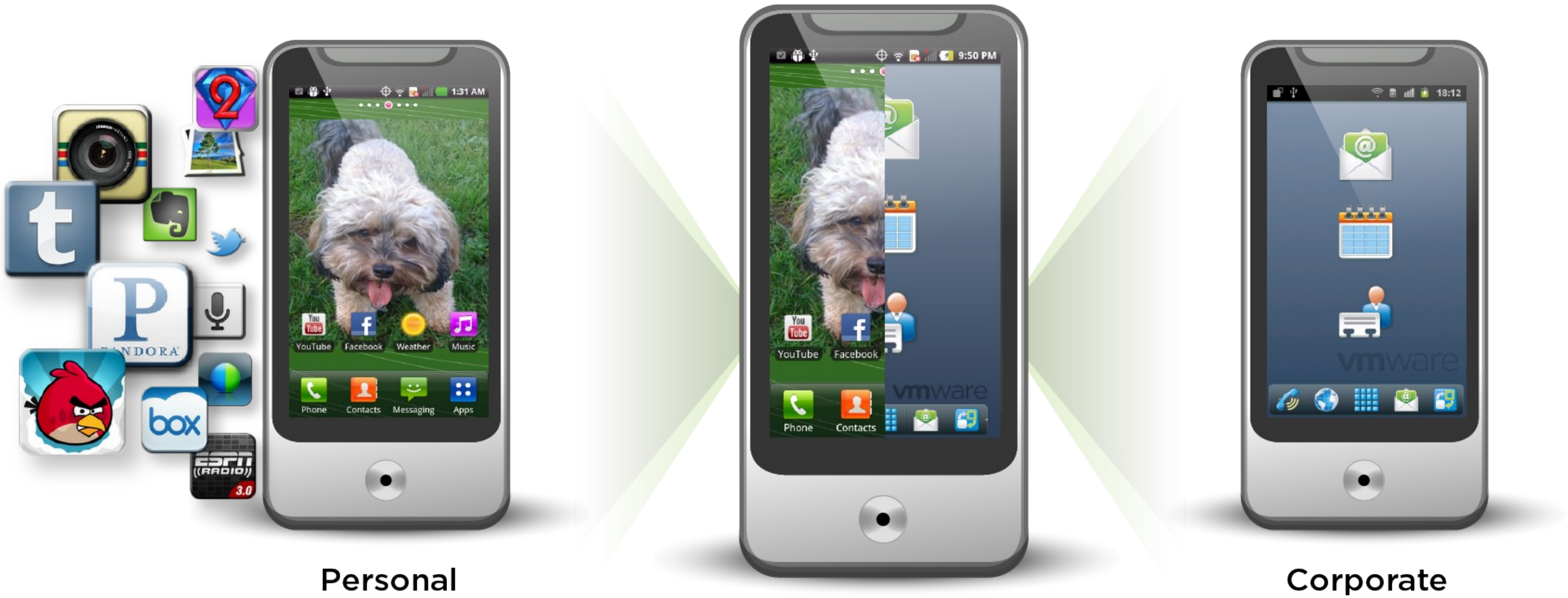
% Virtualized

(of all installed server workloads world wide)



Source: IDC Worldwide Virtualization Tracker, 2010

Not Just Servers



Personal

Corporate

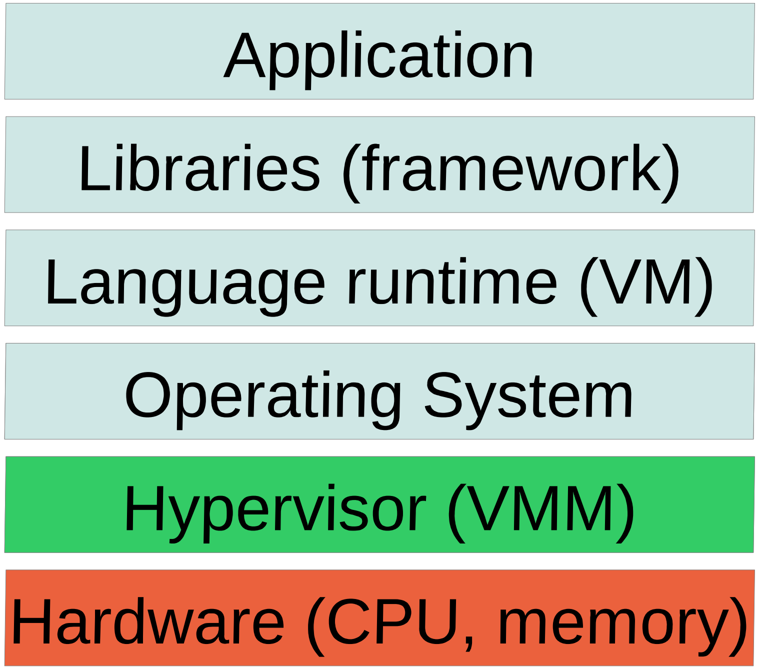
Farming, too

441,137
Add Coins & Cash
FV 21
21752
25
[redacted] Farm
You are visiting [redacted] Farm
Return Home
FUEL

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:

	native	virtual
countPrimes	15.0	15.6
getppid	6.5	29.9



Application

Libraries (framework)

Language runtime (VM)

Operating System

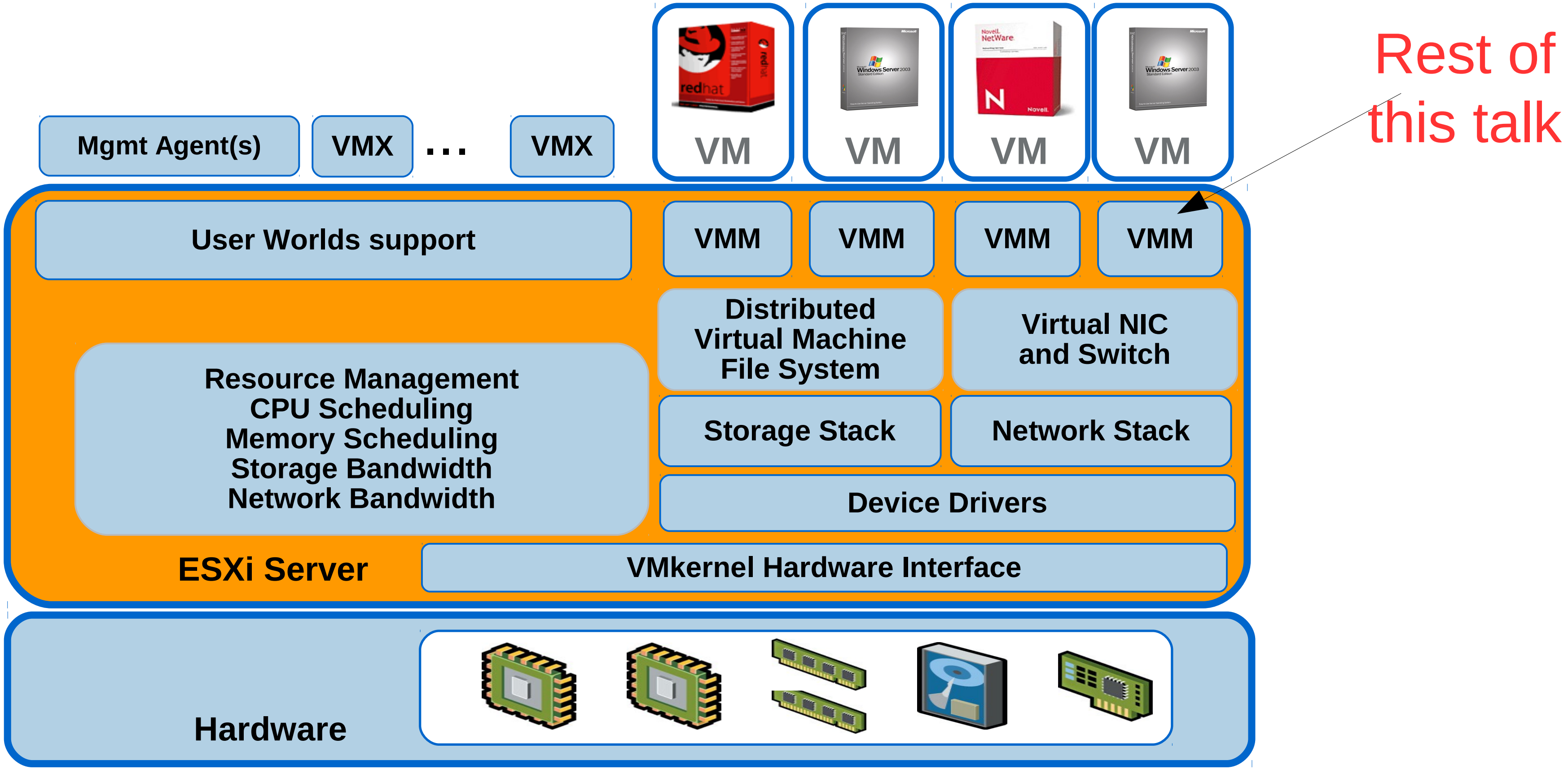
Hypervisor (VMM)

Hardware (CPU, memory)

Outline

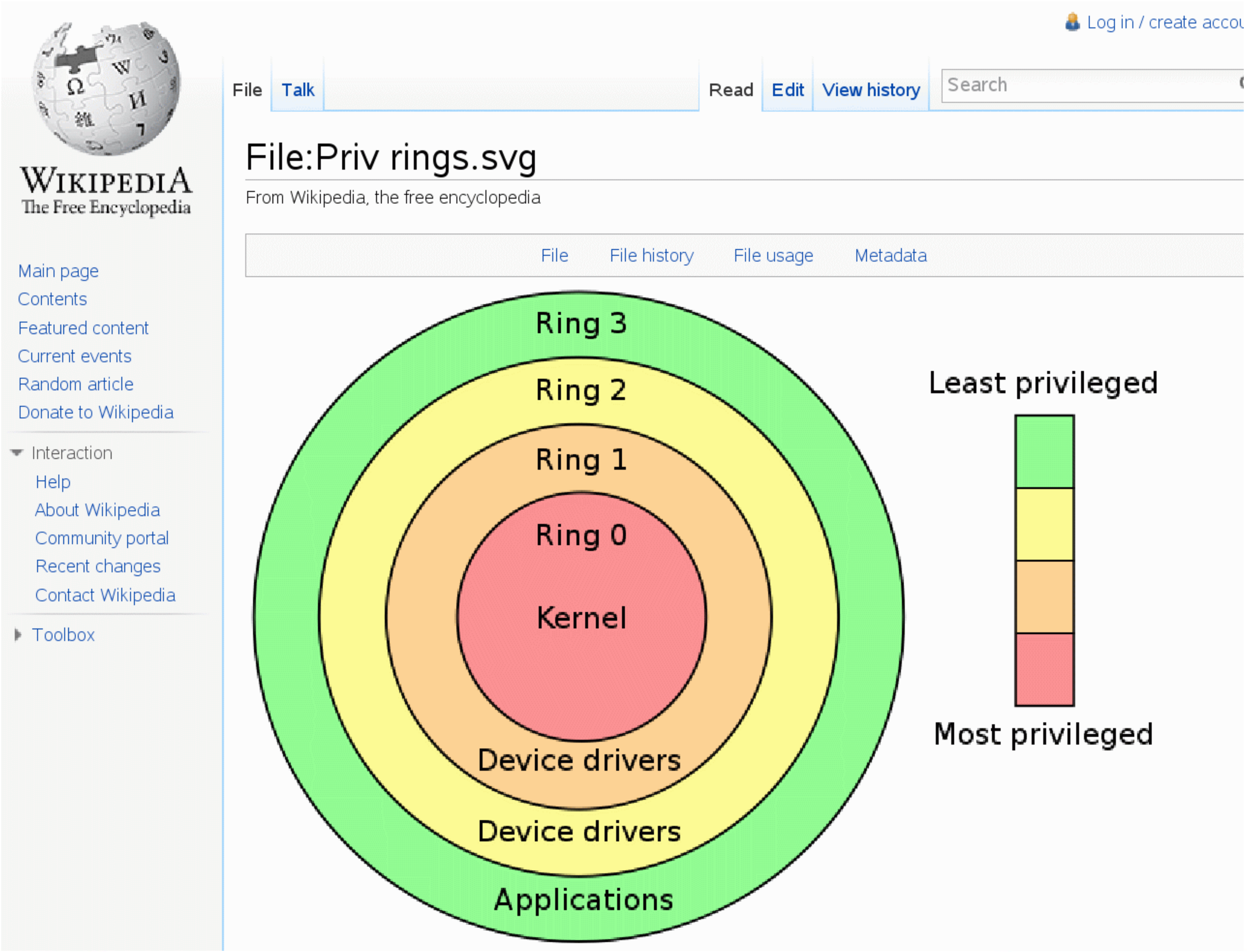
- **Virtualization primer**
- **Instruction set virtualization**
  - Software: binary translation (*BT*)
  - Hardware: Intel VT-x, AMD-V (*HV*)
- **Memory virtualization**
  - Software: shadow page tables
  - Hardware: AMD RVI, Intel EPT (NPT)
- **And beyond . . .**
- **Conclusions**

The Virtual Machine Monitor (VMM) in Context



IBM
370

Protection Rings (CPL = Current Privilege Level)

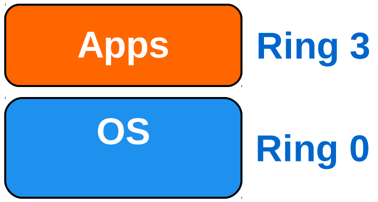


File:Priv rings.svg

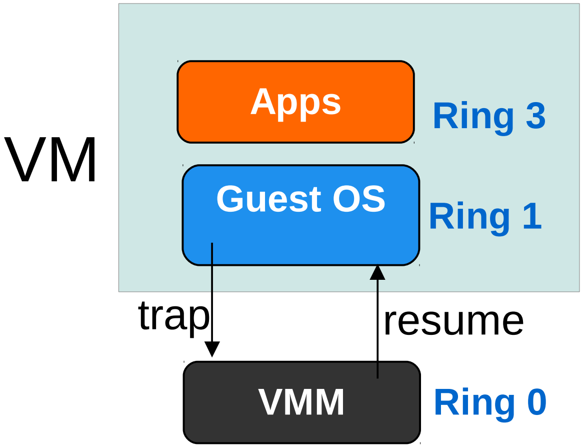
From Wikipedia, the free encyclopedia

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)



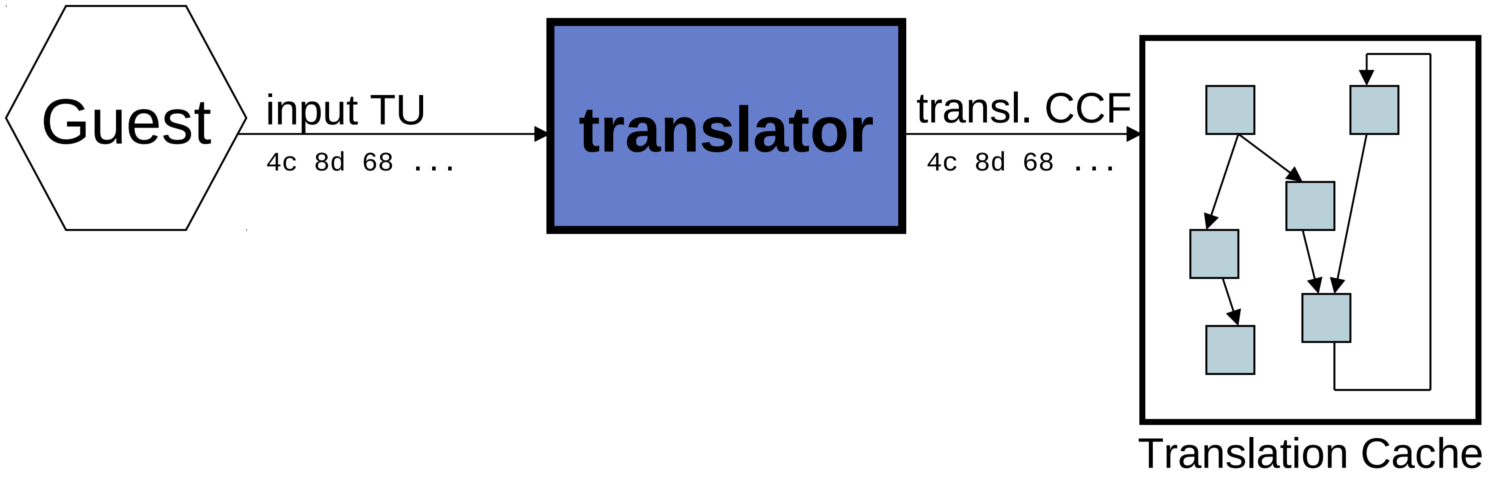
Classical VM Performance

- **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.

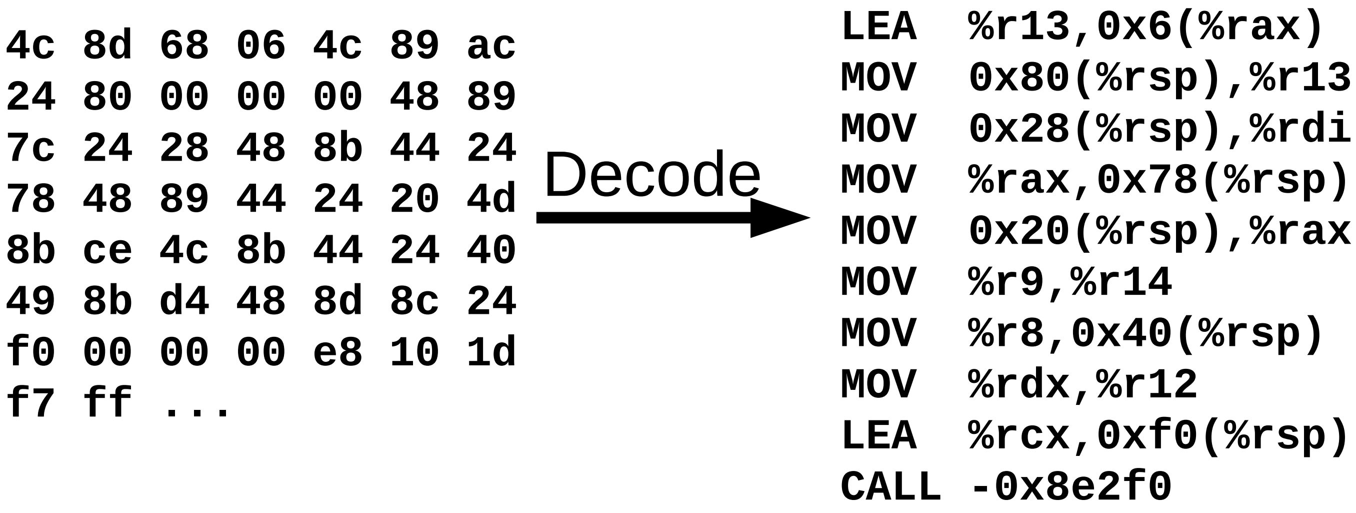
BT Mechanics



- **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 ...
```

```
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 “memcopy”**
- **No slowdown**

TU

```
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
```

→

CCF

```
0xffffffffe681368     LEA   %r13,0x6(%rax)
0xffffffffe68136c     MOV   0x80(%rsp),%r13
0xffffffffe681374     MOV   0x28(%rsp),%rdi
0xffffffffe681379     MOV   %rax,0x78(%rsp)
0xffffffffe68137e     MOV   0x20(%rsp),%rax
0xffffffffe681383     MOV   %r9,%r14
0xffffffffe681386     MOV   %r8,0x40(%rsp)
0xffffffffe68138b     MOV   %rdx,%r12
0xffffffffe68138e     LEA   %rcx,0xf0(%rsp)
0xffffffffe681396     CALL Translation
```

Translation: CALL

- **Push return address on stack**

TU

```
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
```

CCF

```
0xfffffffffe681368     LEA   %r13,0x6(%rax)
0xfffffffffe68136c     MOV   0x80(%rsp),%r13
0xfffffffffe681374     MOV   0x28(%rsp),%rdi
0xfffffffffe681379     MOV   %rax,0x78(%rsp)
0xfffffffffe68137e     MOV   0x20(%rsp),%rax
0xfffffffffe681383     MOV   %r9,%r14
0xfffffffffe681386     MOV   %r8,0x40(%rsp)
0xfffffffffe68138b     MOV   %rdx,%r12
0xfffffffffe68138e     LEA   %rcx,0xf0(%rsp)
0xfffffffffe681396     MOV   %r11,$0x7ff7feda360
0xfffffffffe6813a0     PUSH %r11   ; return address
0xfffffffffe6813a2 <GS>MOV -0x25c0ead(%rip),$0x...
0xfffffffffe6813ad     DISPATCH offs=0xfff71d15
```

Translation: CALL

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

TU

```
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
```

CCF

```
0xfffffffffe681368     LEA   %r13,0x6(%rax)
0xfffffffffe68136c     MOV   0x80(%rsp),%r13
0xfffffffffe681374     MOV   0x28(%rsp),%rdi
0xfffffffffe681379     MOV   %rax,0x78(%rsp)
0xfffffffffe68137e     MOV   0x20(%rsp),%rax
0xfffffffffe681383     MOV   %r9,%r14
0xfffffffffe681386     MOV   %r8,0x40(%rsp)
0xfffffffffe68138b     MOV   %rdx,%r12
0xfffffffffe68138e     LEA   %rcx,0xf0(%rsp)
0xfffffffffe681396     MOV   %r11,$0x7ff7feda360
0xfffffffffe6813a0     PUSH  %r11  ; return address
0xfffffffffe6813a2 <GS>MOV  -0x25c0ead(%rip),$0x...
0xfffffffffe6813ad     DISPATCH offs=0xfff71d15
```

Translation: CALL

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

TU

```
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
```

CCF

```
0xffffffffe681368     LEA   %r13,0x6(%rax)
0xffffffffe68136c     MOV   0x80(%rsp),%r13
0xffffffffe681374     MOV   0x28(%rsp),%rdi
0xffffffffe681379     MOV   %rax,0x78(%rsp)
0xffffffffe68137e     MOV   0x20(%rsp),%rax
0xffffffffe681383     MOV   %r9,%r14
0xffffffffe681386     MOV   %r8,0x40(%rsp)
0xffffffffe68138b     MOV   %rdx,%r12
0xffffffffe68138e     LEA   %rcx,0xf0(%rsp)
0xffffffffe681396     MOV   %r11,$0x7ff7feda360
0xffffffffe6813a0     PUSH  %r11   ; return address
0xffffffffe6813a2 <GS>MOV  -0x25c0ead(%rip),$0x...
0xffffffffe6813ad     DISPATCH offs=0xfff71d15
```

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`)

TU

```
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
```

→

CCF

```
0xfffffffffe681368     LEA   %r13,0x6(%rax)
0xfffffffffe68136c     MOV   0x80(%rsp),%r13
0xfffffffffe681374     MOV   0x28(%rsp),%rdi
0xfffffffffe681379     MOV   %rax,0x78(%rsp)
0xfffffffffe68137e     MOV   0x20(%rsp),%rax
0xfffffffffe681383     MOV   %r9,%r14
0xfffffffffe681386     MOV   %r8,0x40(%rsp)
0xfffffffffe68138b     MOV   %rdx,%r12
0xfffffffffe68138e     LEA   %rcx,0xf0(%rsp)
0xfffffffffe681396     MOV   %r11,$0x7ff7feda360
0xfffffffffe6813a0     PUSH  %r11   ; return address
0xfffffffffe6813a2 <GS>MOV -0x25c0ead(%rip),$0x...
```

CALLEE TRANSLATION...

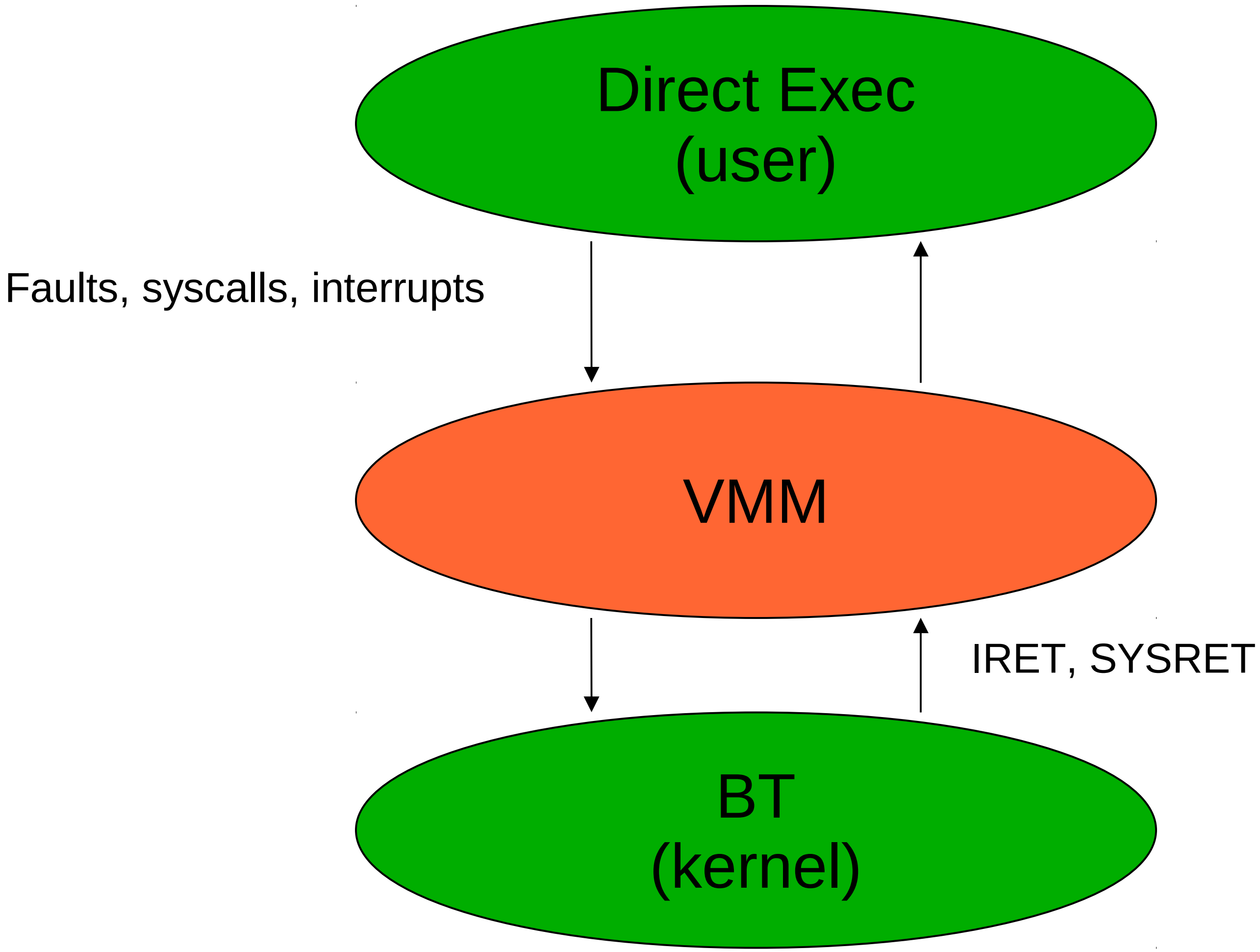
Non-IDENT Translations

- 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

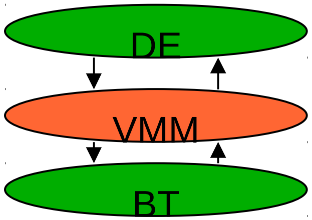
- **Binary**
  - Input is x86 “hex” not source
- **Widening**
  - Output is always 64 bit
  - Input can be real mode, v8086 mode, 16/32/64 bit
- **Dynamic**
  - Interleave translation and execution
- **On-demand**
  - Translate only what we are about to execute
- **System-level**
  - Make no assumptions 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



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**

```
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);
```

	native	DE	BT
countPrimes	15.0	15.6	15.7

Performance examples for BT

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

`fib(46) = 2971215073`

```
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));
```

	native	DE	BT
countPrimes	15.0	15.6	15.7
fib(46)	21.3	21.1	54.7

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)

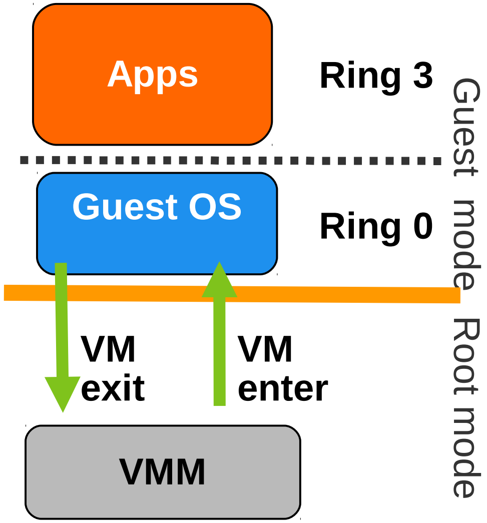
```
for (i = 0; i < 10000000000; i++) {
  getppid();
}
```

	native	DE	BT
countPrimes	15.0	15.6	15.7
fib(46)	21.3	21.1	54.7
getppid	6.5	29.9	

- **Microbenchmarks**
  - Not directly representative of real workloads (“undiluted evilness”)

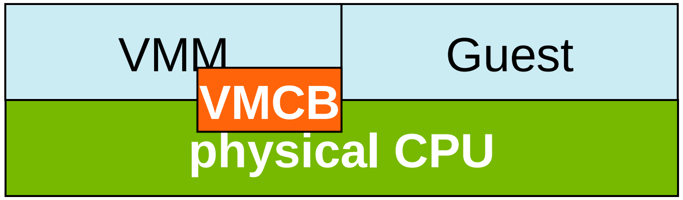
Using Hardware Support (HV) to Run Guests (Intel VT-x, AMD-V)

- **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**

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

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

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**

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

	native	BT	HV
getppid	6.5	29.9	6.8

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();
}
```

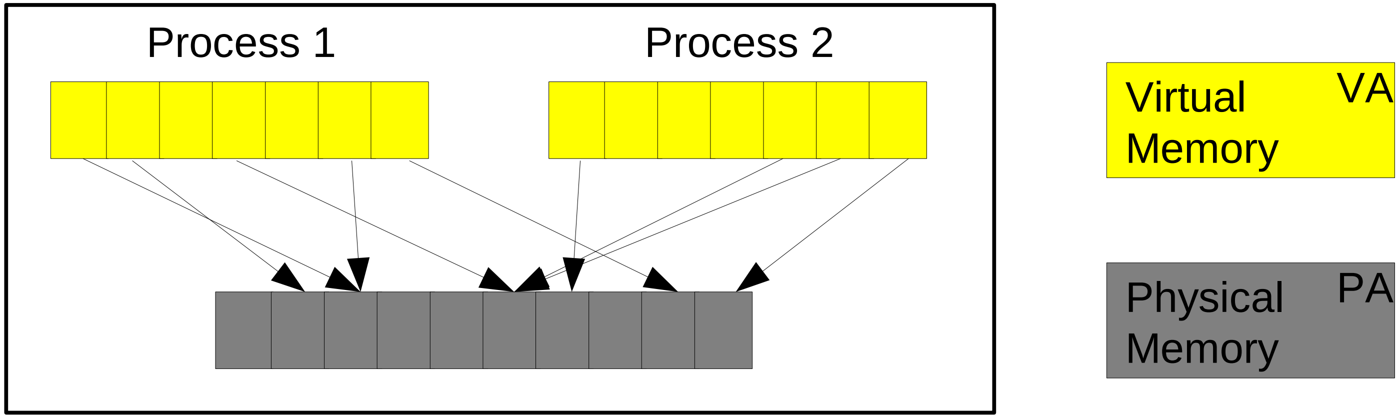
	native	BT	HV
getppid	13.8	114	14.9
forkwait	5.4	22.8	24.3

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

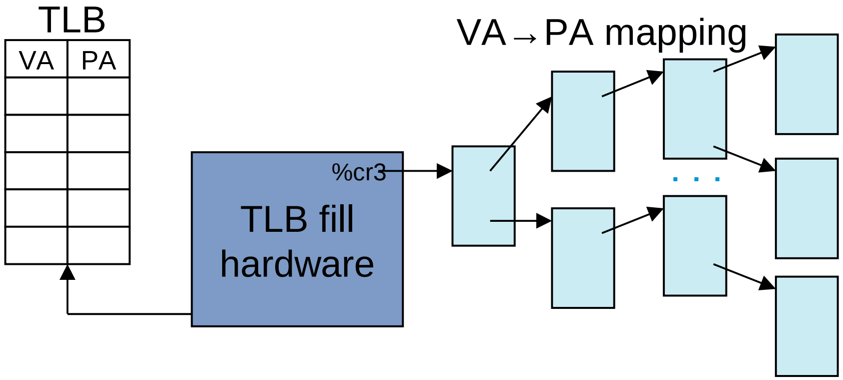
Memory Virtualization

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

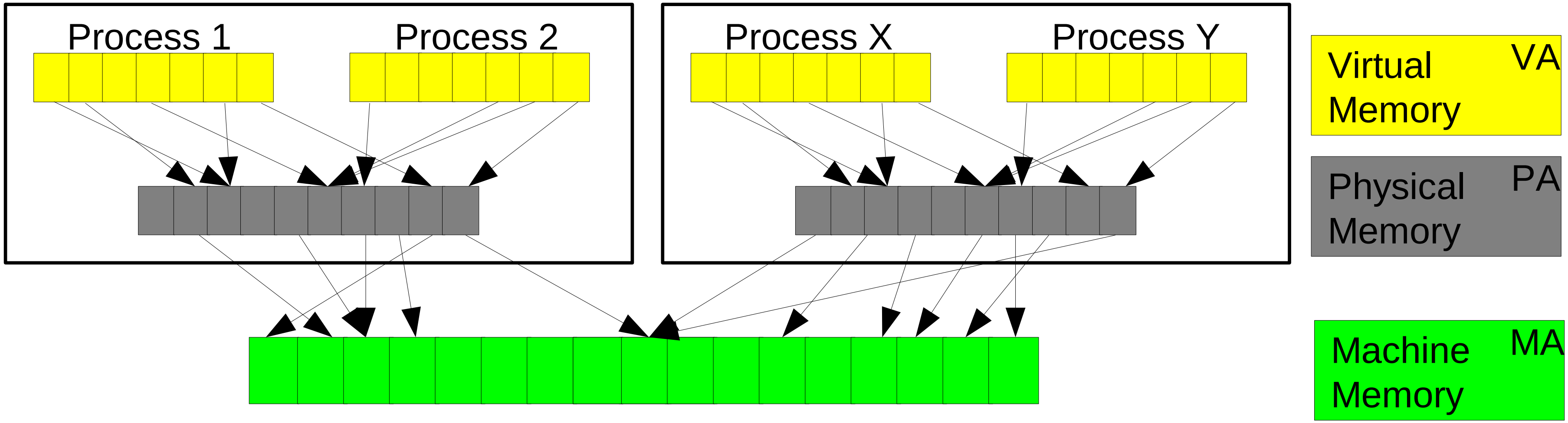
Virtual Memory



- **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)

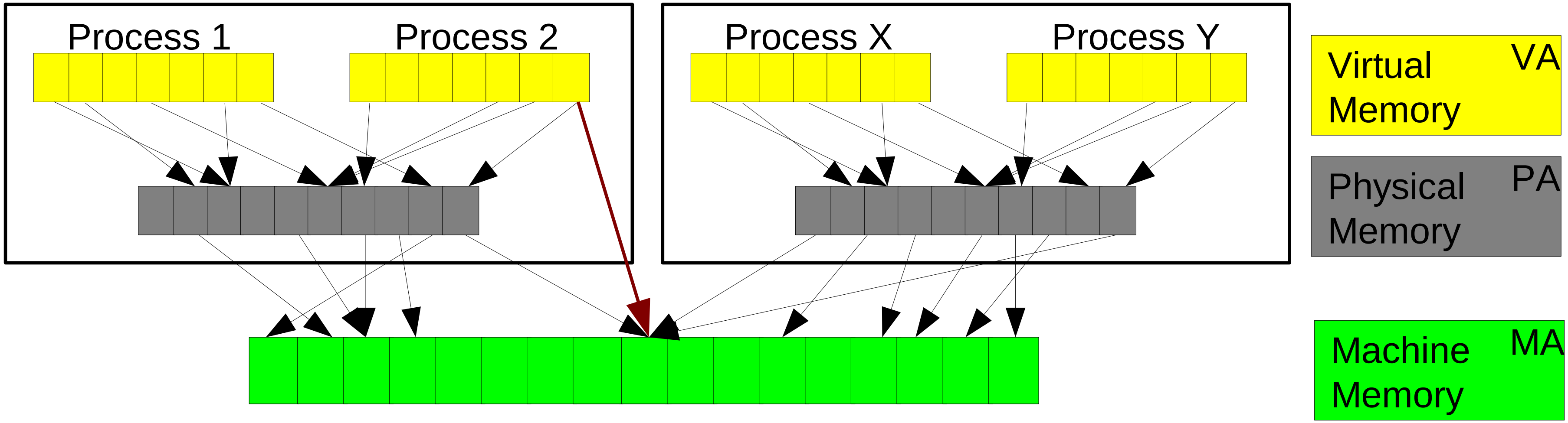


Virtualizing Virtual Memory



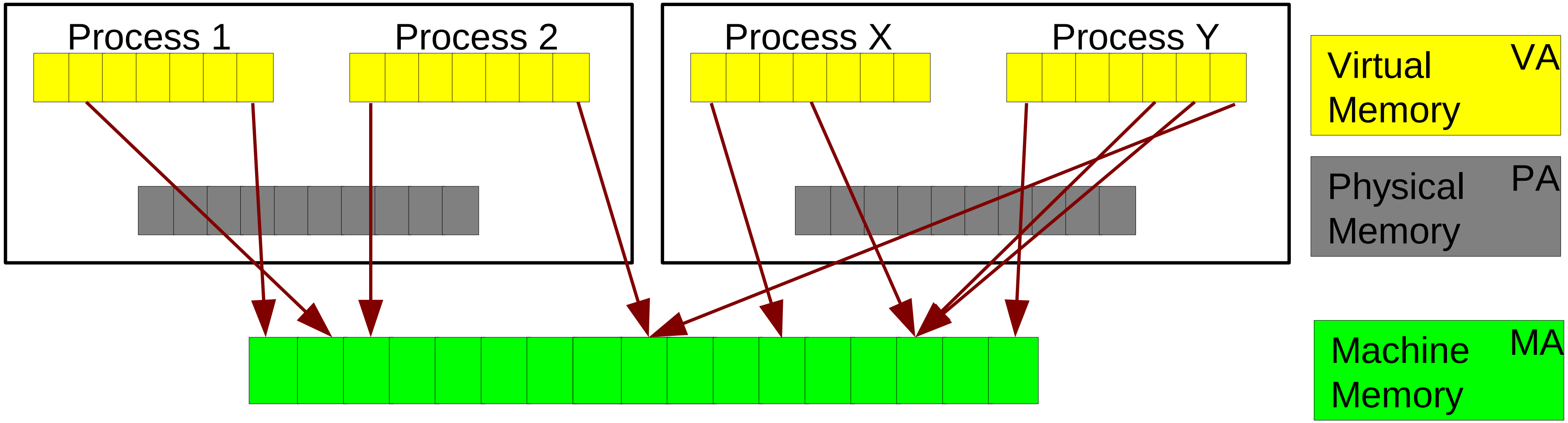
- **To run multiple VMs, another level of memory indirection is needed**
  - Guest OS controls virtual to physical mapping: VA→PA
    - No access to machine memory (to enfore isolation)
  - VMM maps guest physical to machine memory: PA→MA
- **How to make this go fast?**

Software Solution: Shadow Page Tables

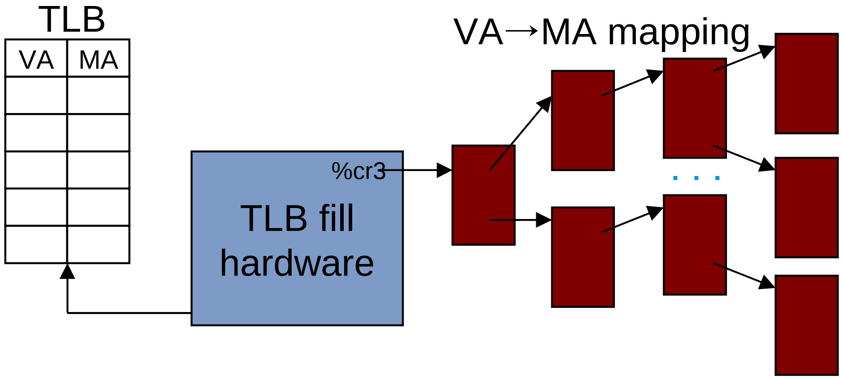


- **VMM builds “shadow page tables” to accelerate the mappings**
  - Composition of VA→PA and PA→MA
  - Shadow directly maps VA→MA

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

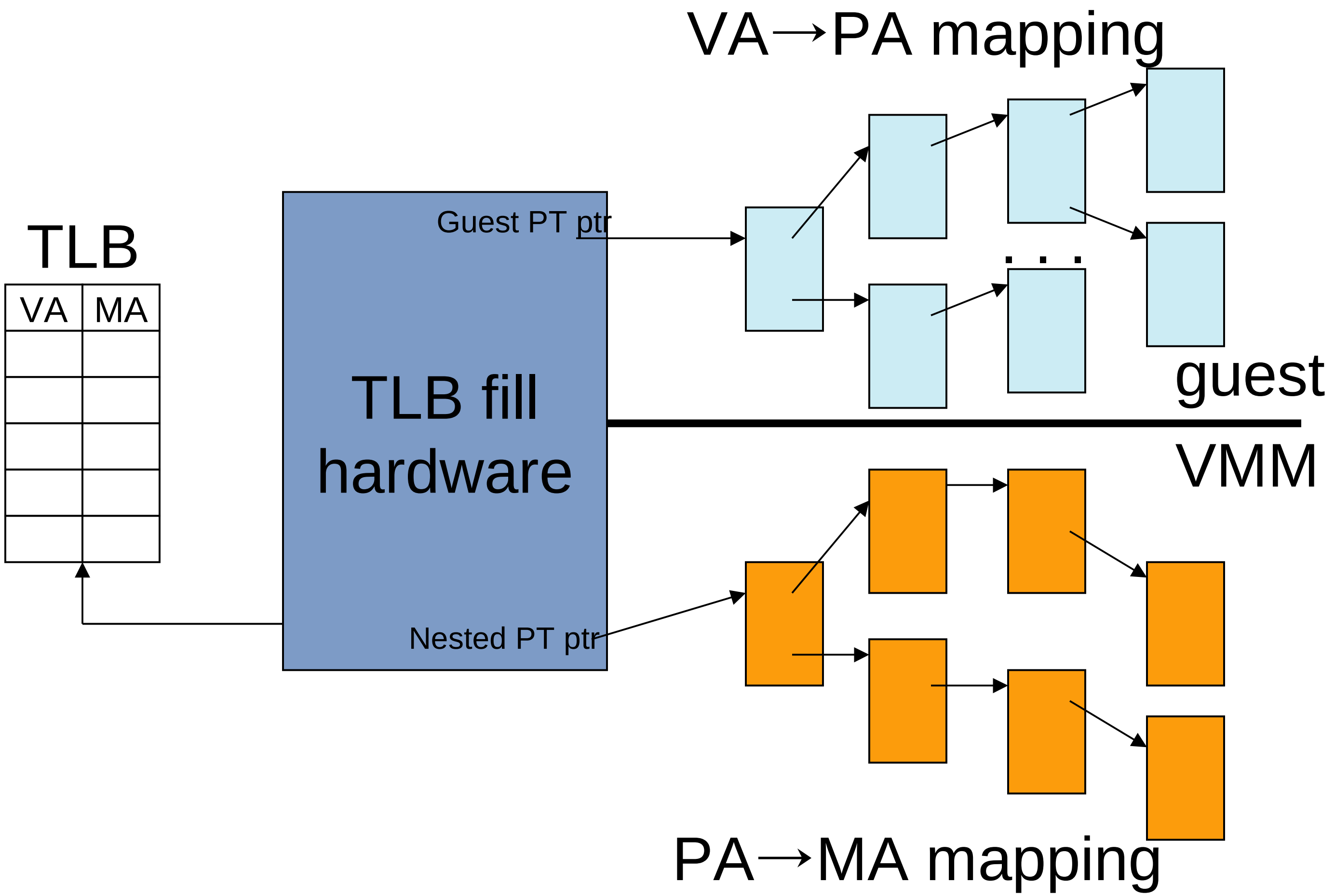
- VMM must intercept page faults
- Validate shadow (hidden page fault), or forward fault to guest (true fault)

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)



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**

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

	native	BT	HV	RVI
getppid	6.5	29.9	6.8	6.8
forkwait	5.4	22.8	24.3	12.6

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

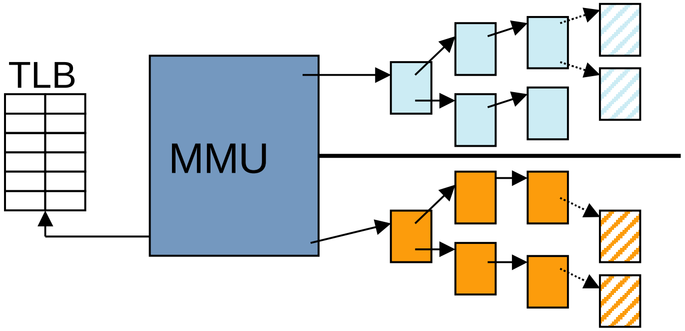
```
#define S  (8192 * 4096)
volatile char large[S];

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

	native	BT	HV	RVI
getppid	6.5	29.9	6.8	6.8
forkwait	5.4	22.8	24.3	12.6
memsweep	12.0	12.8	12.8	26.5

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

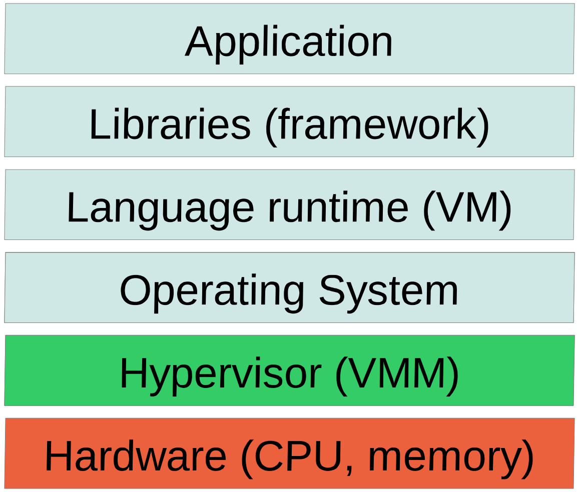
- **CPUs have hardware-based event counters**
  - For performance analysis (and debugging)
  - Tools like vtune and perf gather and visualize data
  - Want such tools to work in VMs
- **Must virtualize performance counters**
  - Let counters run through exits?
  - Stop counters at exit?
  - Best solution 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

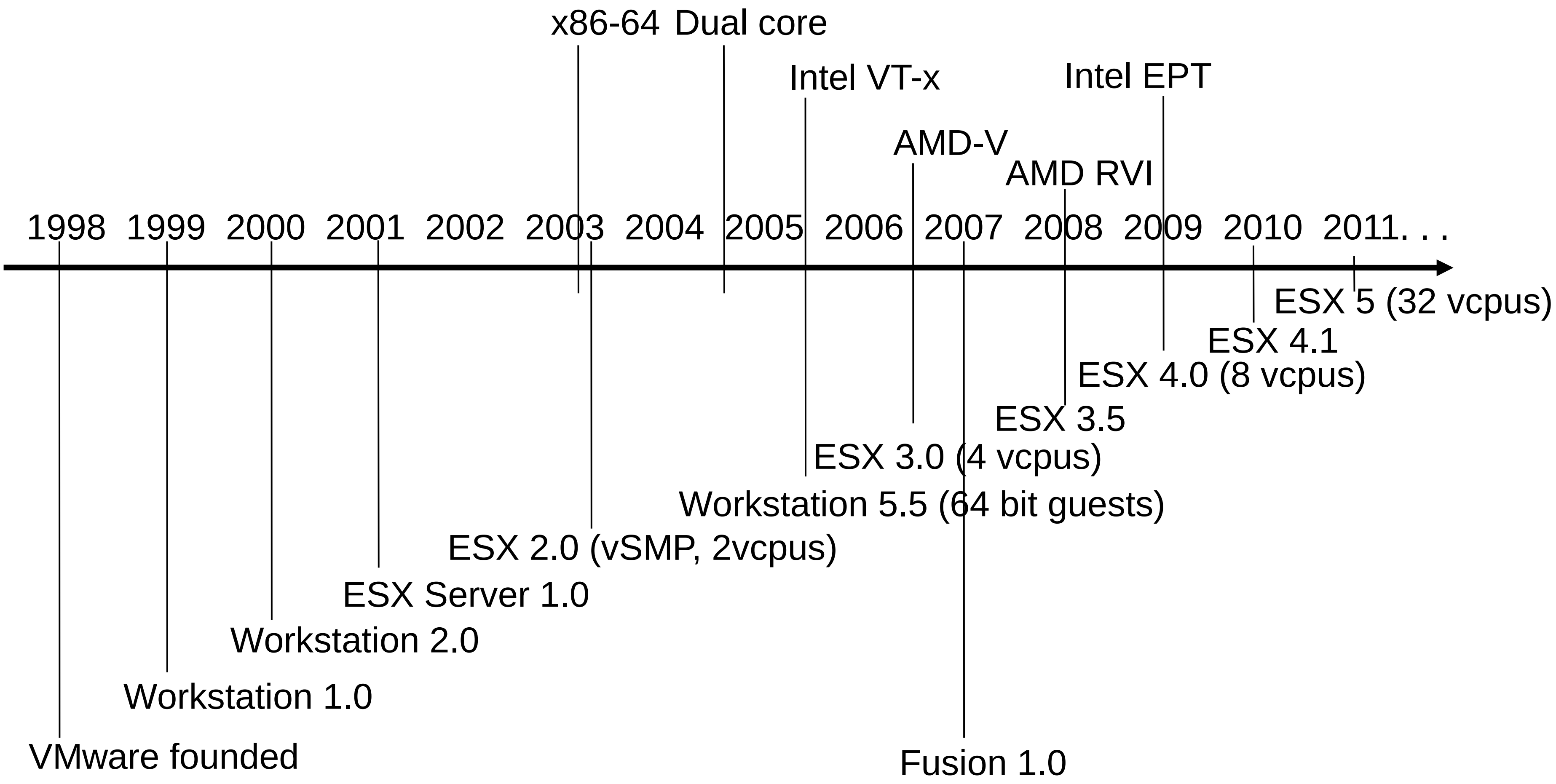
- **Adams, Agesen. *A Comparison of Software and Hardware Techniques for x86 Virtualization*. ASPLOS 2006.**
- **Agesen, Garthwaite, Sheldon, Subrahmanyam. *The Evolution of an x86 Virtual Machine Monitor*. Operating Systems Review, 44(4), 2010.**
- **Agesen, Mattson, Rugina, Sheldon. *Software Techniques for Avoiding Hardware Virtualization Exits*. ATC 2012.**
- **Ben-Yehuda, Day, Dubitzky, Factor, Har'El, Gordon, Liguori, Wasserman, Yassour. *The Turtles Project: Design and Implementation of Nested Virtualization*. OSDI 2010.**
- **Bhargava, Serebrin, Spadini, Manne. *Accelerating Two-Dimensional Page Walks for Virtualized Systems*. ASPLOS 2007.**
- **Neiger, Leung, Rodgers, Uhlig. *Intel Virtualization Technology: Hardware support for Efficient Processor Virtualization*. Intel Technology Journal 10, 3. 2006.**
- **Scales, Nelson, Venkitachalam. *The Design of a Practical System for Fault-Tolerant Virtual Machines*. Operating Systems Review 44(4), 2010.**
- **Serebrin, Hecht. *Virtualizing Performance Counters*. Workshop on System-level Virtualization for High Performance Computing, EuroPar 2011.**
- **Zhang, Garthwaite, Baskakov, Barr. *Fast Restore of Checkpointed Memory Using Working Set Estimation*. VEE 2011.**

Backup slides

xenon.stanford.edu

```
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
   This is Xenon.Stanford.EDU
       4x Intel(R) Xeon(R) X5650 @2.67Hz, 4GB RAM, CentOS 5.x x86_64
       Please report Xenon problems to: action@xenon
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
-> Xenon's Acceptable Use Policy is at http://cs.stanford.edu/aup/xenon

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



x86-64 Dual core

Intel VT-x

Intel EPT

AMD-V

AMD RVI

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

ESX 5 (32 vcpus)

ESX 4.1

ESX 4.0 (8 vcpus)

ESX 3.5

ESX 3.0 (4 vcpus)

Workstation 5.5 (64 bit guests)

ESX 2.0 (vSMP, 2vcpus)

ESX Server 1.0

Workstation 2.0

Workstation 1.0

VMware founded

Fusion 1.0

Windows 2000 Boot/Halt Translation Stats

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

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+%**