# A Tale of 10 Bugs: Performance Engineering at VMware

*Ravi Soundararajan, SB ‘92 (VMware, Inc.)*

*MIT Guest Lecture, 6.172*

*12/9/10*

# A Case Study in Performance Engineering

## Email thread from a colleague

*"…*

*Interestingly, as the number of <benchmark> threads decreased hostsPerThread var increases), the percentage of locktime spent in dbwrites also increases.*

*…*

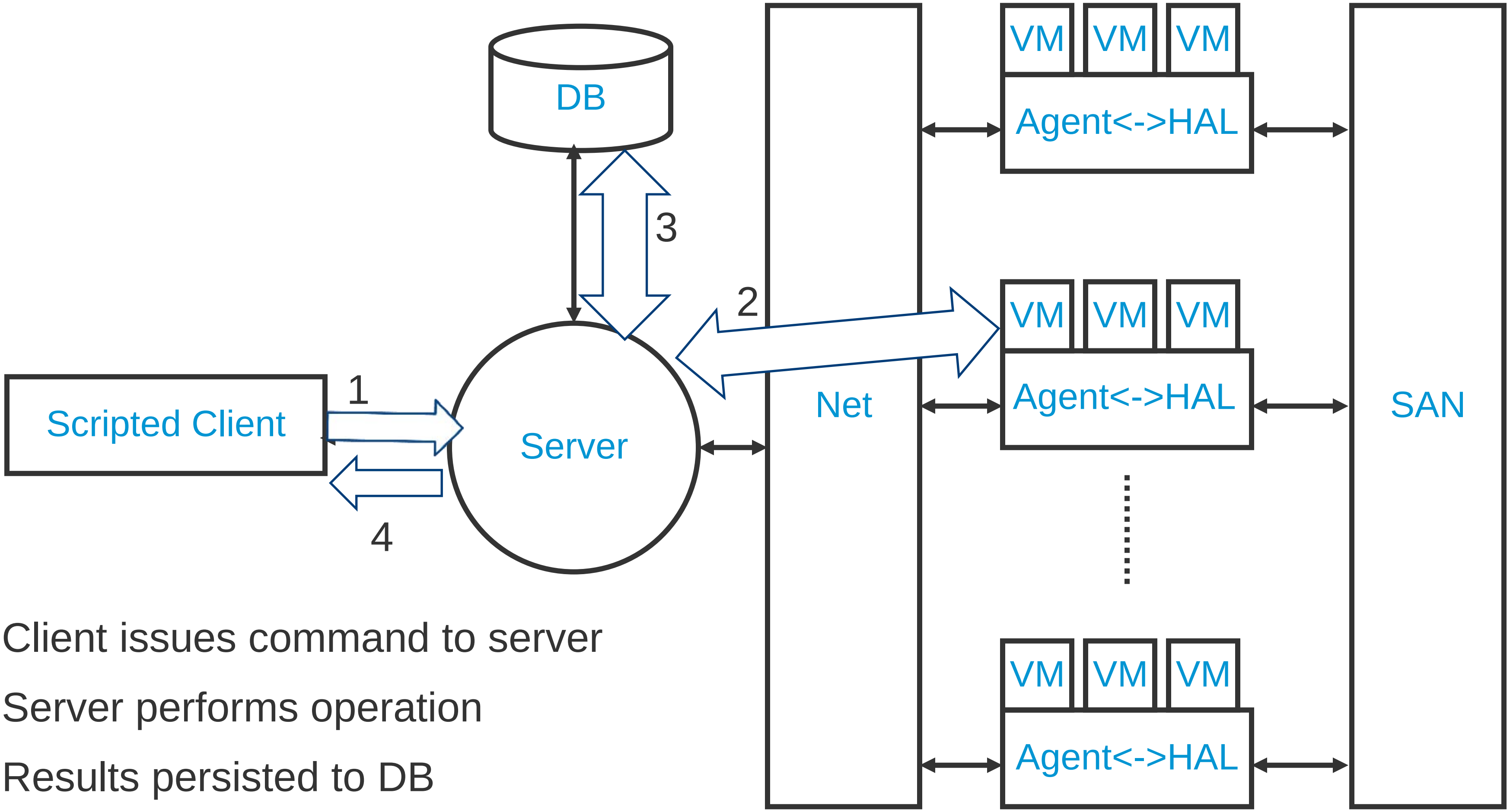
*lots of threads (hostsPerThread = 4):*

- *~28 % lock time spent under vdbWrite Connection*
- *~16 % lock time spent under exec / commit.*

*…"*

Translation: Why is % lock time in DB increasing despite lighter load?

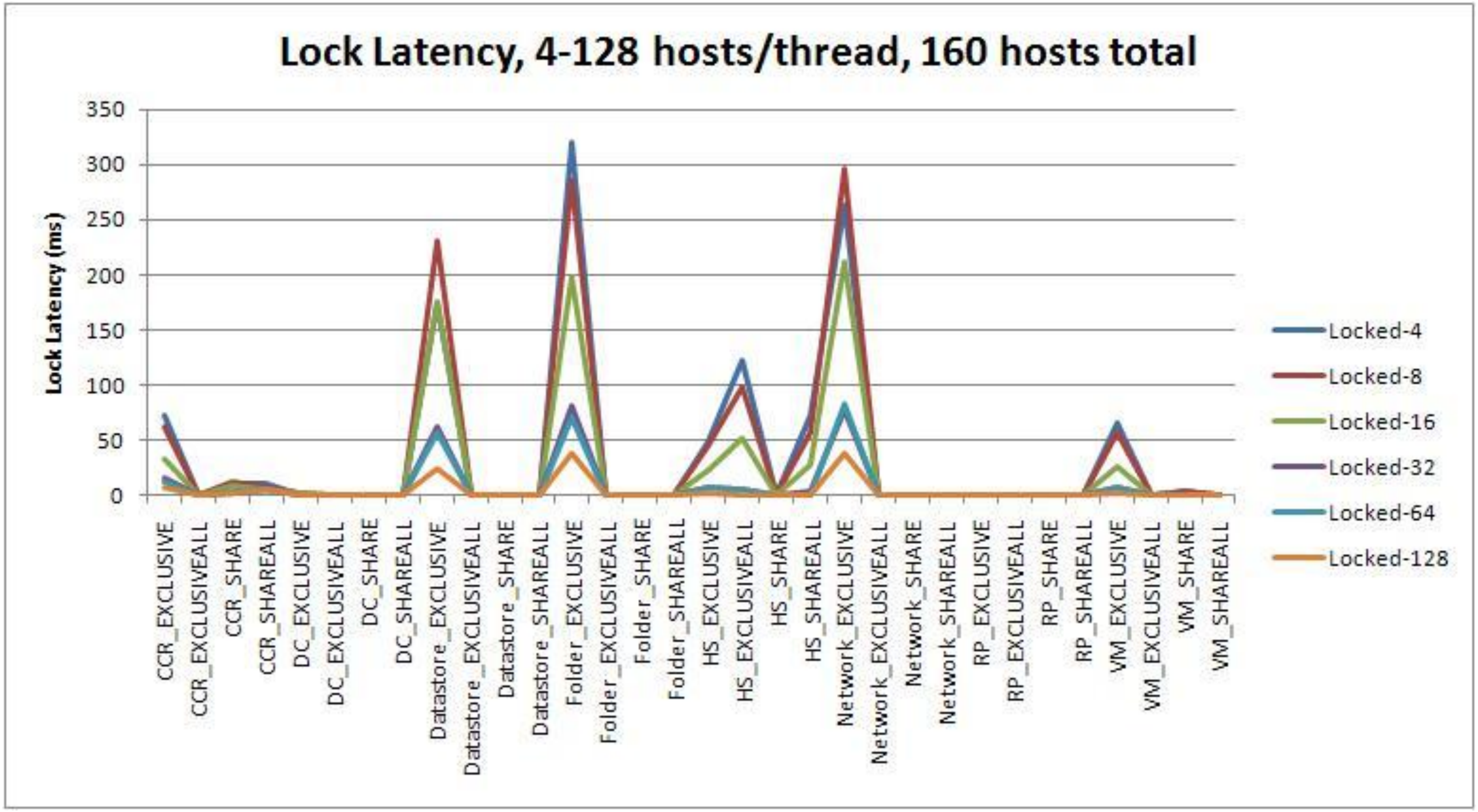
# Step 0: What the ?%##!* is he talking about?




1. Client issues command to server
2. Server performs operation
3. Results persisted to DB
4. Client is notified of completion

Problem: With ***lighter*** load from client, %time spent in DB Locks ***increases***

# Step 1: Examine Lock Hold Time for Various Loads

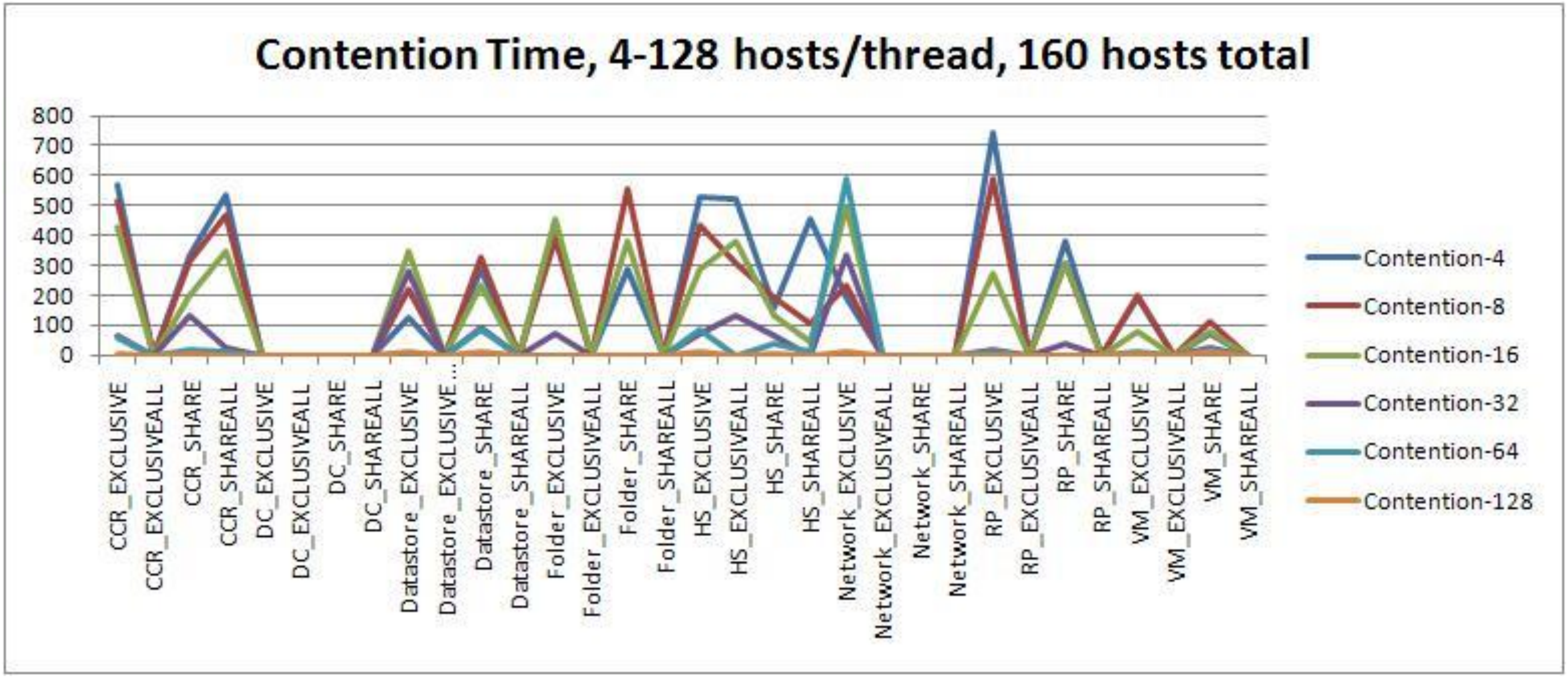



Latency per lock @ 128 hosts/thread < 4 hosts/thread (Expected…lighter load)

→Original question: why is %DB increasing with lighter load?

→Answer: DB latency dominates when overall latency is lower!

# Step 2: Examine Contention Time for Various Loads




Contention per lock @ 128 hosts/thread < 4 hosts/thread (OK…lighter load)

→With lighter load, less overall contention time and higher % of time @ DB

# *Post mortem* on Case Study #1

1. Understand experimental setup (multi-tier setup)

2. Understand what is being measured (% time in DB lock)

3. Examine relevant data (lock latency)

4. Draw appropriate conclusion
   - Yes, % lock time in DB is higher with a lighter load
   - BUT, overall lock time is small with lighter load
   - Therefore, DB lock time (roughly constant) contributes more to lock latency

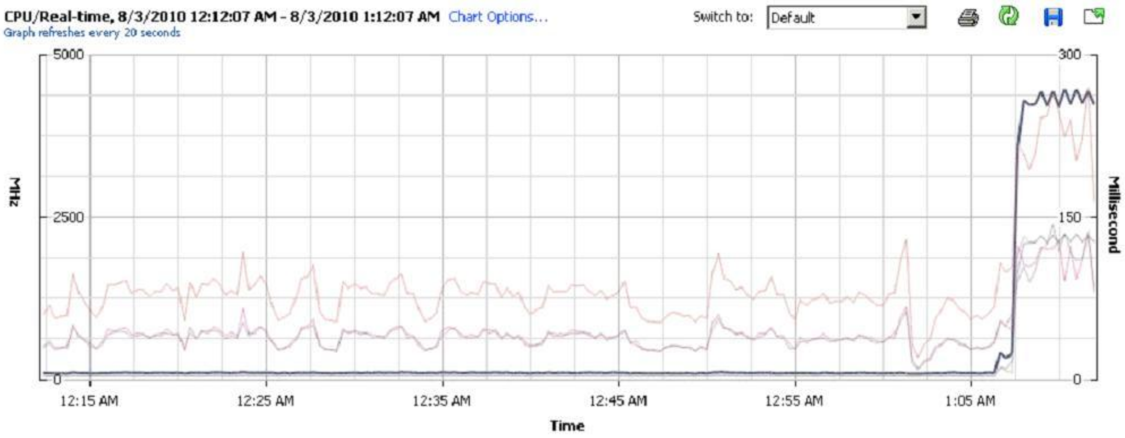
# Outline

**Case Studies in Performance Engineering @ VMware**

Lessons Learned

# Case Study #2: Garbage In, Garbage Out

## Customer wants to draw this chart:




**Performance Chart Legend**

| Key | Object | Measurement | Rollup | Units | Latest | Maximum | Minimum | Average |
|---|---|---|---|---|---|---|---|---|
| | SpecJBB-esx1 | CPU Usage in MHz | Average | MHz | 4237 | 4472 | 78 | 421.883 |
| | SpecJBB-esx1 | CPU Ready | Summation | Millisecond | 165 | 269 | 20 | 87.061 |
| | 0 | CPU Ready | Summation | Millisecond | 85 | 132 | 10 | 43.789 |
| | 1 | CPU Ready | Summation | Millisecond | 81 | 143 | 9 | 43.272 |
| | 0 | CPU Usage in MHz | Average | MHz | 2119 | 2233 | 40 | 210.611 |
| | 1 | CPU Usage in MHz | Average | MHz | 2102 | 2221 | 34 | 207.15 |

## PowerCLI

- CPU Usage for a VM for last hour:
- $vm = Get-VM –Name “Foo”
- Get-Stat –Entity $vm –Realtime –Maxsample 180 –Stat cpu.usagemhz.average
- Grab appropriate fields from output, use graphing program, etc.

# What Happens at Scale? Comparing PowerCLI and Java

| Entities (cpu.usagemhz.average) | PowerCLI (Time in secs) | Java (Time in secs) |
|---|---|---|
| 1 VM | 9.2 | 14 |
| 6 VMs | 11 | 14.5 |
| 39 VMs | 101 | 16 |
| 363 VMs | 2580 (43 minutes) | 50 |

Highly-tuned Java Stats Collector

A Naïve script that works for small environments may not be suitable for large environments

Translation: Garbage In, Garbage Out…*but why?*

# PowerCLI vs. Java

## PowerCLI

- Toolkit: meant for ease of use…hides details
- Similar to a shell script: facilitates quick prototyping
- Stateless

## Java

- Harder to use
- But…can use more advanced techniques (data structures, thread pools, etc.)

# What's going on behind the scenes?

This is what is going on ***for each Get-Stat call in PowerCLI***

- Retrieve PerformanceManager
- QueryPerfProviderSummary $vm → ***Says what intervals are supported***
- QueryAvailablePerfMetric $vm → ***Describes available metrics***
- QueryPerfCounter → ***Verbose description of counters***
- Create PerfQuerySpec → ***Query specification to get the stats***
- QueryPerf → ***Get stats***

***Bottom line: The PowerCLI toolkit spares you details…Easy to use!***

# Optimizing the Java Code

PowerCLI

```
Get VM ID
for each Get-Stat {
        QueryAvailablePerfMetric();
        QueryPerfCounter();
        QueryPerfProviderSummary();
        create PerfQuerySpec();
        QueryPerf();
}
```

Java

perfCounter property
Of
PerformanceManager

Get VM ID

~~QueryAvailablePerfMetric();~~

~~QueryPerfCounter();~~

```
QueryPerfProviderSummary();
create PerfQuerySpec();
for each Get-Stat {
        QueryPerf();
}
```

PowerCLI: 5 RPC calls per VM. Java: 1 RPC call per VM.

Further optimization not shown: Java allows more compact format

# Why Garbage In, Garbage out?

## PowerCLI

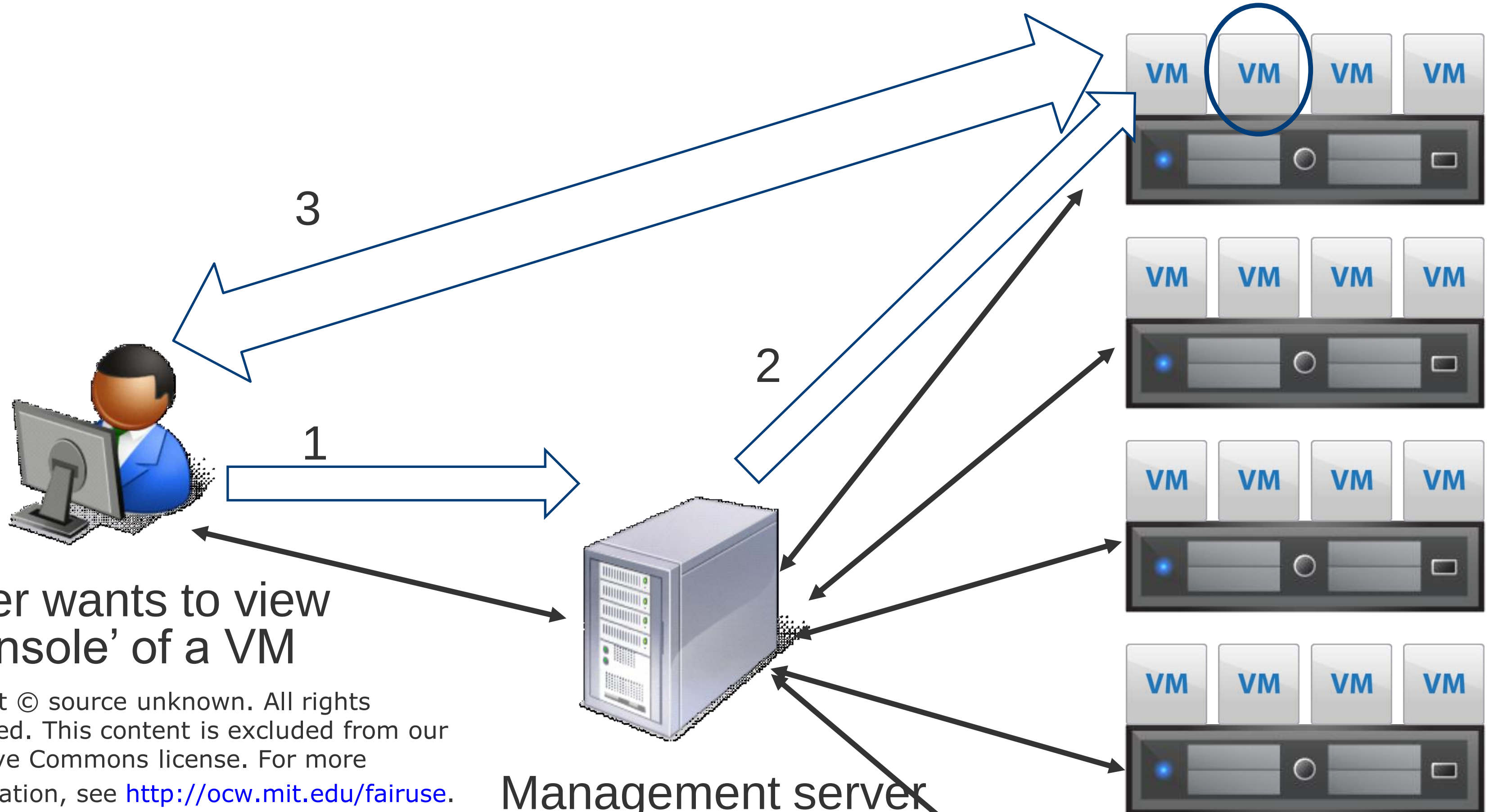
- Wrote a ‘simple’ but non-optimized script
- Did not utilize multi-threading (split up VM list, use muliple client queries)
- Did not realize output format is verbose
- Did not realize # of RPC calls is 5*O(#VMs)

## Java

- Utilized multiple threads
- Understood what data was the same across VMs → reduce redundant calls
- Utilized more compact output format (CSV vs. raw objects)
- Reduced # of RPC calls

*(Think about assembly code vs. compiler-generated code)*

# Case Study #3: A Lesson in API Design






1. User talks to management server
2. Management server locates VM
3. User & VM get connected

# The Problem: Remote Console Doesn't Show Up!

- **Problem: could not start VM remote console in large environment**

- **Sequence of debugging**
  - Client folks: it's a server problem
  - Server folks: it's a client problem
  - Client folks: it's a 'vmrc' problem (vmrc = VMware Remote Console)
  - VMRC folks: authentication? MKS tickets?
  - Me: this is ridiculous…

- **More Information: Start remote console for a single VM**
  - 50 Hosts, no problem
  - 500 Hosts, no problem
  - 1001 Hosts, PROBLEM!

# No Console: Examining the Cases the Actually  Work

- **Debugging observations**
  - With < 1000 hosts…
    - Management server CPU and memory goes very high when client invoked
    - Console is dark until CPU and memory go down, then appears
  - Look at server log file
    - Data retrieval call occurs before console appears (WHY???)
    - In failure cases, exception in serializer code
  - Attach debugger
    - Exception is an out-of-memory exception
    - Exception is silently ignored (never returns to client)

# No Console: Isolating the Problem

- **Problem**
  - VMRC creates a request to monitor host information (e.g., is CD-ROM attached)
  - Request gets info on ALL hosts
  - At 1001 hosts, we exceed 200MB buffer on server
  - 200MB restriction only for old-style API clients

- **Solution**
  - VMRC folks: do NOT create big request
  - Server folks: fail correctly and emit better errors

***Lessons***

1. ***Create APIs that are difficult to abuse, rather than easy to abuse***
2. ***Teach clients how to use APIs***
3. ***Make sure (internal) users have input about API design***

# Case Study #4: 32-bit vs. 64-bit (Thanks, R. M.!)

**Benchmark run**

- Build A: 100 ops/min.
- Build B: 50 ops/min.

**What was the difference?**

- Build A: 32-bit executable on 64-bit hardware
- Build B: 64-bit executable on 64-bit hardware

**Huh?**

# 4 (b) xPerf

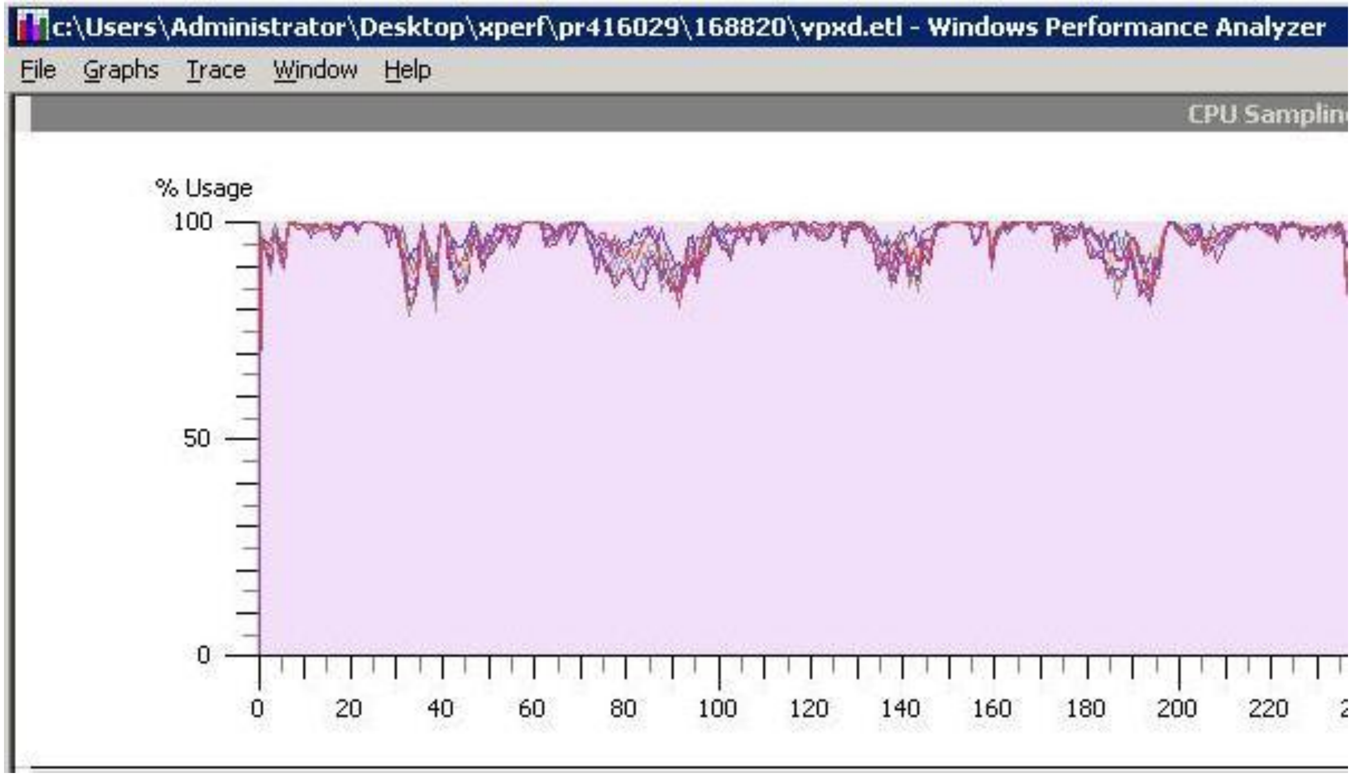
Runs on Windows 2008

Sampling profiler (with other cool attributes)

Records stack traces

Give caller/callee information

# 4 (c) CPU Saturation in 64-bit case




CPU is mostly saturated (in 32-bit case, CPU is not saturated)

# 4(d) Look at Sampling Profile

| Stack | Weight | % Weight |
|---|---|---|
| | 3,532,612.638 ... | 87.39 |
| [Root] | 3,532,455.583 ... | 87.38 |
| \|- ntdll.dll!RtlUserThreadStart | 3,508,983.437 ... | 86.80 |
| \|- ntdll.dll!ZwQueryVirtualMemory | 13,762.103 969 | 0.34 |
| \|- ntkrnlmp.exe!KiSystemServiceCopyEnd | 5,903.542 252 | 0.15 |
| \|- ntkrnlmp.exe!KiChainedDispatch | 901.654 720 | 0.02 |
| \|- ntkrnlmp.exe!KiDpcInterrupt | 321.063 214 | 0.01 |
| \|- ntdll.dll! ?? ::FNODOBFM::`string' | 258.007 162 | 0.01 |
| \|- ntdll.dll!RtlImageNtHeaderEx | 226.071 622 | 0.01 |
| \|- MSVCR80.DLL!_RTtypeid | 105.035 557 | 0.00 |

Shows stacks originating from root

Shows 87% CPU used from 1 process

But this is just the thread start routine, where threads originate

# 4(e) The Perils of Sampling Profilers

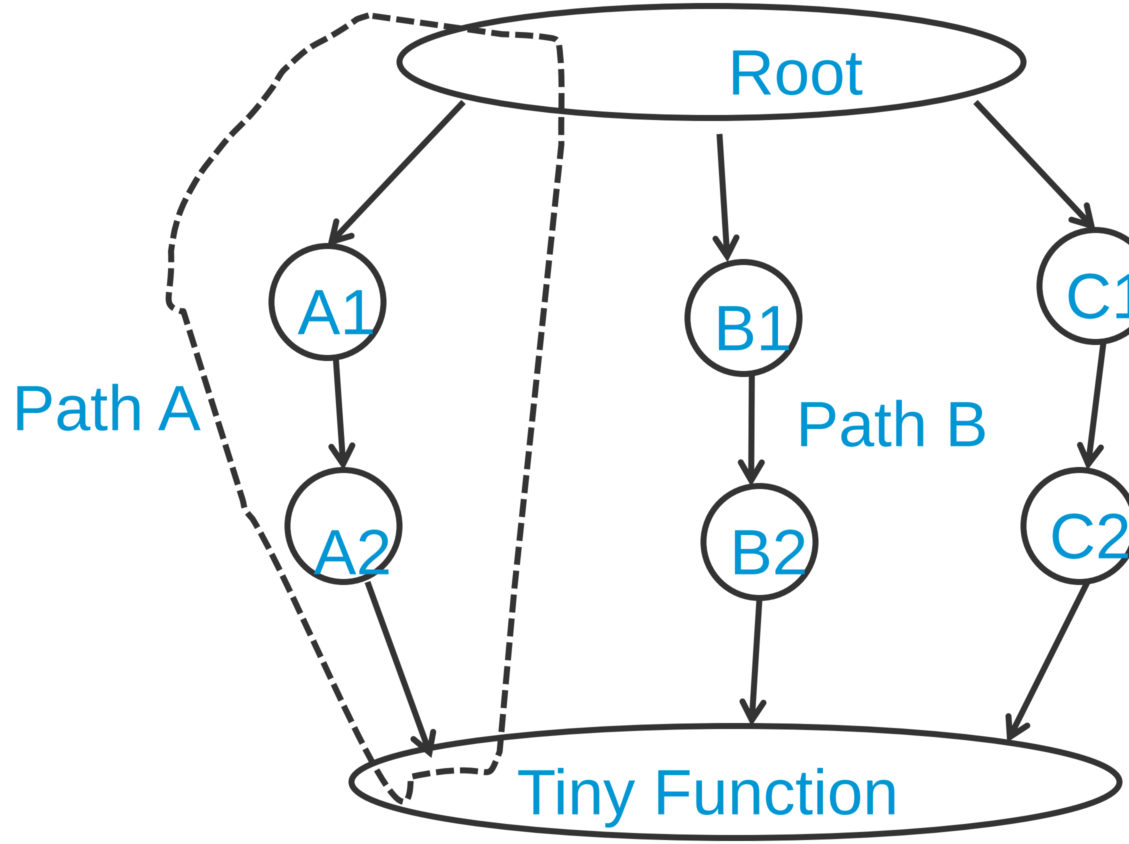
| Stack | Weight | % Weight |
|---|---|---|
| | 3,532,612.638 ... | 87.39 |
| [Root] | 3,532,455.583 ... | 87.38 |
| ntdll.dll!RtlUserThreadStart | 3,508,983.437 ... | 86.80 |
| kernel32.dll!BaseThreadInitThunk | 3,508,983.437 ... | 86.80 |
| vpxd.exe!Win32ThreadMain | 2,270,619.910 ... | 56.17 |
| vpxd.exe!VpxdThread::ThreadFunc | 2,270,619.910 ... | 56.17 |
| vpxd.exe!VpxLroList::ThreadMainEntry | 2,045,997.133 ... | 50.61 |
| vpxd.exe!VpxLRO::LroMain | 2,041,093.903 ... | 50.49 |
| vpxd.exe!VpxActivationLRO::InvokeA... | 1,571,975.867 ... | 38.89 |
| vmomi.dll!Vmomi::ManagedMethod... | 1,518,822.158 ... | 37.57 |
| vmomi.dll!sVmodlQueryPropert... | 770,053.167 788 | 19.05 |
| types.dll!sVimVirtualMachineDi... | 727,687.116 523 | 18.00 |
| vpxd.exe!VpxdMoVm::Po... | 672,630.405 844 | 16.64 |

From Root, most of the samples are from this call stack

Most popular stack, but is this the problem?

# 4(f) Perils of Sampling Profilers, Part 2

**Most-common trace: not necessarily where time is spent**




Many paths to “Tiny Function’

Maybe time spent here?

# 4(g) The Caller View

## Look at Callers for various routines in stacks

| Callers | Weight | % Weight |
|---|---|---|
| ntdll.dll!ZwQueryVirtualMemory | 3,123,003.752 ... | 77.26 |
| \|- ntdll.dll! ?? ::FNODOBFM::`string' | 3,109,241.648 ... | 76.92 |
| \| \|- MSVCR80.DLL!_RTDynamicCast | 1,579,519.929 ... | 39.07 |
| \| \|- MSVCR80.DLL!_RTtypeid | 1,529,451.706 ... | 37.84 |
| \| \|- [Root] | 257.005 987 | 0.01 |
| \| \|- MSVCR80.DLL!CxxThrowException | 13.006 609 | 0.00 |
| \|- [Root] | 13,762.103 969 | 0.34 |
| ntdll.dll!ZwQueryVirtualMemory | 2.004 444 | 0.00 |

Not called a lot from root, however…

Called from few places and takes 77% CPU!

RTtypeid?



vmware

# 4(h) RTtypeid?

| Callers | Weight | % Weight |
|---|---|---|
| ⊟ MSVCR80.DLL!_RTtypeid | 1,583,683.660 ... | 39.18 |
| ⊞ \|- vmacore.dll!Vmacore::ObjectImpl::IncRef | 828,866.361 590 | 20.50 |
| ⊞ \|- vmacore.dll!Vmacore::ObjectImpl::DecRef | 725,473.308 729 | 17.95 |
| ⊞ \|- MSVCR80.DLL!_RTDynamicCast | 28,925.004 534 | 0.72 |
| ⊞ \|- vpxd.exe!ManagedObjectMapper::operator() | 283.942 793 | 0.01 |
| ⊞ \|- [Root] | 105.035 557 | 0.00 |
| ⊞ \|- vpxd.exe!VpxLRO::GetStatsContext | 18.005 567 | 0.00 |
| ⊞ \|- vpxd.exe!DrmModule::SnapshotDomain | 10.999 304 | 0.00 |
| ⊞ \|- vmacore.dll!Vmacore::PrintFormatter::FormatException | 1.002 432 | 0.00 |

Hmm. RTtypeid is used in figuring out C++ type

39% of overall CPU?

IncRef and DecRef are main callers

# 4(i) The Offending Code

```
void
ObjectImpl::IncRef()
{
   if (_refCount.ReadInc() == 0) {
      const type_info& tinfo = typeid(*this);
      FirstIncRef(tinfo);
   }

    …
}
```

Dynamic cast…needs run-time type info (RTTI)

RTTI has pointers in it

# 4(j) But Why is 64-bit slower than 32-bit?

**Runtime type info (RTTI) has a bunch of pointers**

- 32-bit: pointers are raw 32-bit pointers

- 64-bit
  - Pointers are 32-bit offsets
  - Offsets must be added to base addr of DLL/EXE in which RTTI resides
  - Result is a true 64-bit pointer

**But wait…why is addition slow?**

# 4(k) Why Is Addition Slow? Well, it isn't…

**Addition isn't slow, but…**

**Determining module base address can be slow**

- To find base address, RTtypeid calls RtlPcToFileHeader
- RtlPcToFileHeader grabs loader lock, walks list of loaded modules to find RTTI data
- This can be slow
- N.B.: This is why we see calls to zwQueryVirtualMemory

***For more info:***

**http://blogs.msdn.com/junfeng/archive/2006/10/17/dynamic-cast-is-slow-in-x64.aspx**

# 4(I) What Did We Learn?

**RtTypeId is called from a bunch of places**

**RtTypeId is not, however, called from Root too often**

**RtTypeId is small and fast: not main contributor in most stacks (*except IncRef and DecRef*)**

**Lots of little calls add up**

**Caller view was important here!**

**(btw: 2 solutions:**

- 1. Statically compute base addr and cache
- 2. Use latest runtime library, which avoids RtlToPcFileHeader)

***Lesson: Little things (32-bit vs. 64-bit) may matter…don't discriminate!***

# Case Study #5: Memory Usage Woes

## Why is excessive memory usage a problem?

- Can slow down application if paging is induced
- May cause application to crash (if you exceed per-process limit…2GB in 32-bit Windows)

## Memory leak vs. memory accumulation

- Leak: memory was allocated, not live anymore (dangling reference)
- Accumulation: pointer exists to data, but data not used anymore (a logical leak)

# Tools for Analyzing Memory Usage

## Windows:

- Purify, GlowCode, Memory Validator, malloc hooks and heap dump utilities from Microsoft, etc.

## Linux:

- Valgrind, malloc hooks from Google (example: http://goog-perftools.sourceforge.net/), etc.

## Basic idea:

- Hook calls to malloc
- Figure out liveness of pointers (do you leave scope without free()?)
- But…can be unusably slow if you do a lot of allocations!

# A Trivial Memory Leak

```
void bar() {
    foo();
}

void foo() {
    char *p = malloc(24);
    <do some computation>
    return; /* memory pointed at by p is never freed */
}
```

# Memory Analysis

## Easing memory allocation in C++: use reference-counted objects instead of “naked” pointers

- Each use of an item increments a reference count
- When no references exist, delete the item
- Does not solve memory accumulation problem

# Memory Performance Problem

## Server application runs out of memory after several hours

## Use Purify (on a much smaller setup):

- Leak not detected because data was assigned to a reference
- Instead, examine memory in use
  - Do 100 iterations of an operation
  - See 6400B of allocations for an item (100 64B allocations)
  - Code inspection revealed that item was actually not used anymore…a “logical” leak

  (i.e., there was a free(), but it was never called because the item was thought to be in use)

*Lesson:*

*If an effect is small, find ways to magnify it.*

# Case  Study #6: Another Memory Analysis Problem

User complains that server is getting slower and slower

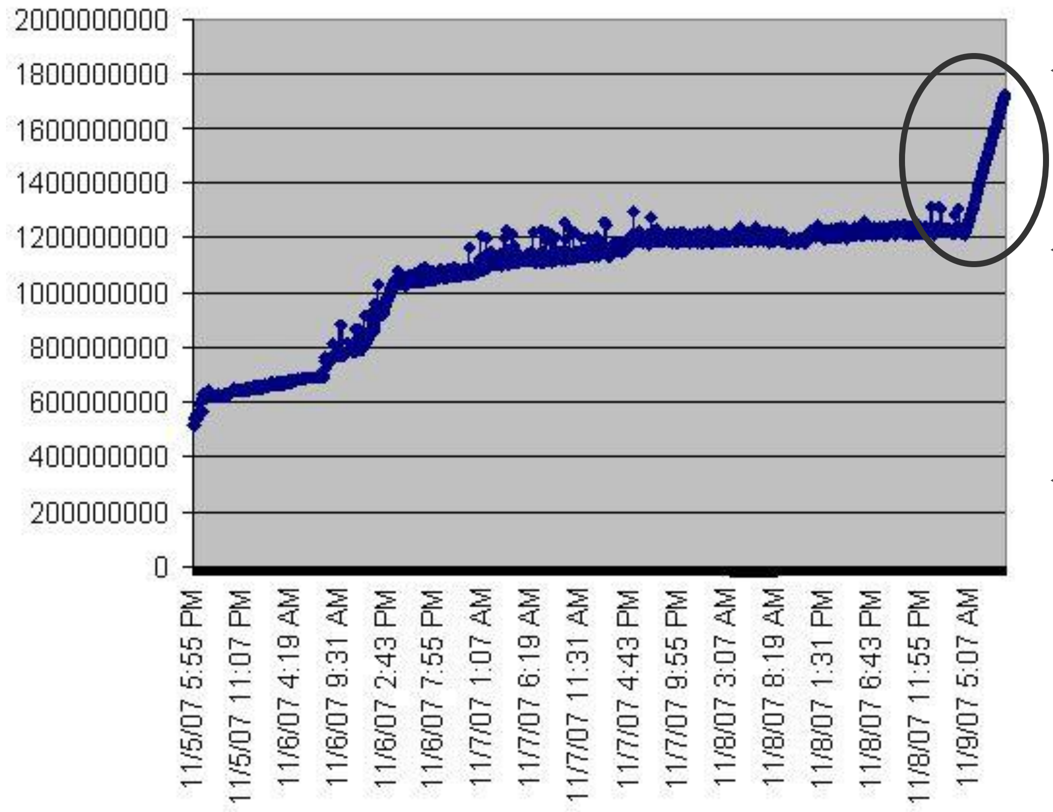
CPU/network/disk not saturated

Memory, however, is increasing dramatically

Eventually, system crashes

# Looking at Memory Usage: Perfmon in Windows

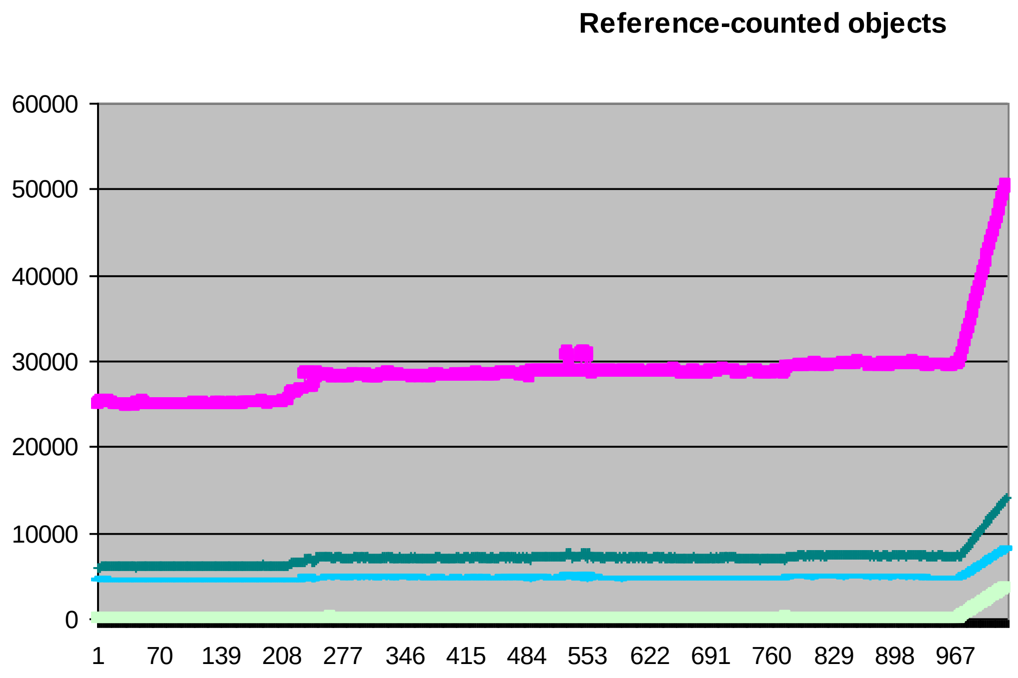
## Chart of "Private Bytes" for a process vs. time




- Memory growing at alarming rate! Not good.
- *Private bytes:* memory committed to process (swap space is allocated for it)
- Memory given by OS to app, not necessarily memory requested by app (example: fragmentation)

**Server is functioning fine, but memory is growing really fast. This could lead to a crash. Let's investigate…**

# Profiling Reference-counted Objects

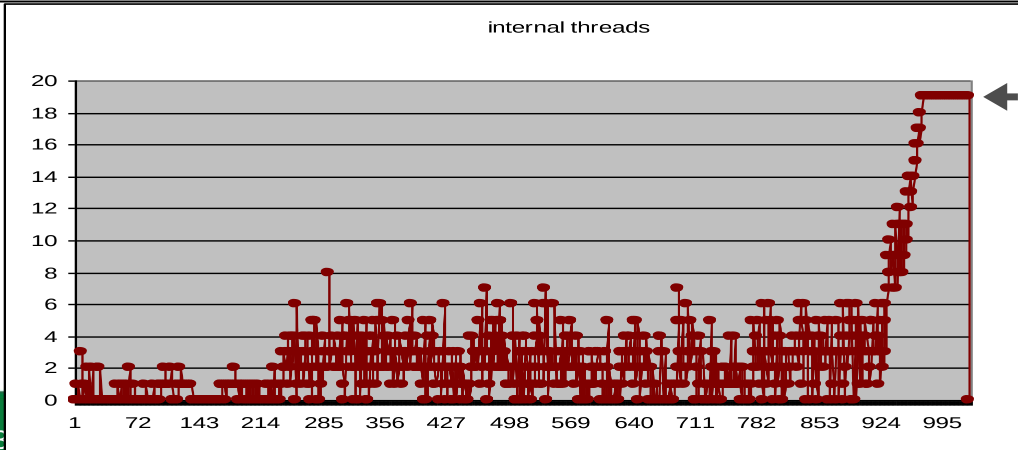



Pink: mutex

Teal: condition variable

Blue: thread activation state

**Some thread-related objects increasing**




**Hmm…number of threads consumed is also increasing**

***** Log files show threads being killed due to uncaught exceptions**

# Customized Profiling: Pros and Cons

## Advantages of our customized profiler:

- Tailored to our application
- Can be made very fast
- Can be run in production environments

## Disadvantages:

- Requires code recompilation (then again, so does Purify)
- Specific to this application (code must be refactored for use in other apps)
- Only counts ref-counted objects: what about C code? What about non-ref-counted objects?

***Lesson:*** Memory profiling is critical.

Sad Reality: Sometimes, commercial tools don’t work at scale

→You may have to write your own

# Case Study #7: How well do you understand networking?

## User issues a request to perform an operation on a VM

- Setup A: Client/Server version 1 to host version 1: 8s
- Setup B: Client/Server version 2 to host version 1: 16s
- Consistent, repeatable difference
- Regression when using new code to talk to older host!

## Step 1: Log everywhere

- Client-imposed latency: same in both cases
- Server-imposd latency: same
- Host imposed-latency: extra 8s in Setup B ➔ Focus on the host

# Networking Issue: Analyzing the host

## Step 2: More logging (standard tools aren't available on host)

- Narrow down the time...
  - Agent <-> HAL, Setup A: 10ms per call
  - Agent <-> HAL, Setup B: 200ms per call
  - Wow!

## Step 3: Examine configuration

- Setup A: named pipe between Agent and HAL
- Setup B: TCP/IP connection between Agent and HAL

# Networking Issue: Resolution

## Step 4: Solution (intuition by developer)

- Named pipe communication, setup A: 10ms
- TCP/IP communication, setup B: 200ms
- Why? Nagle algorithm on socket connection
  - On a TCP socket, wait for more data before sending packets
  - Can be disabled through TCP_NODELAY option

## Step 5: Result

- Use TCP_NODELAY, both have same performance
- Eventually use a cache to avoid interprocess communication

## Lesson?

- “Little” changes can mean a lot
- Client/server code: understand the client/server interaction!

# Case Study #8: Correctness Impacts Performance

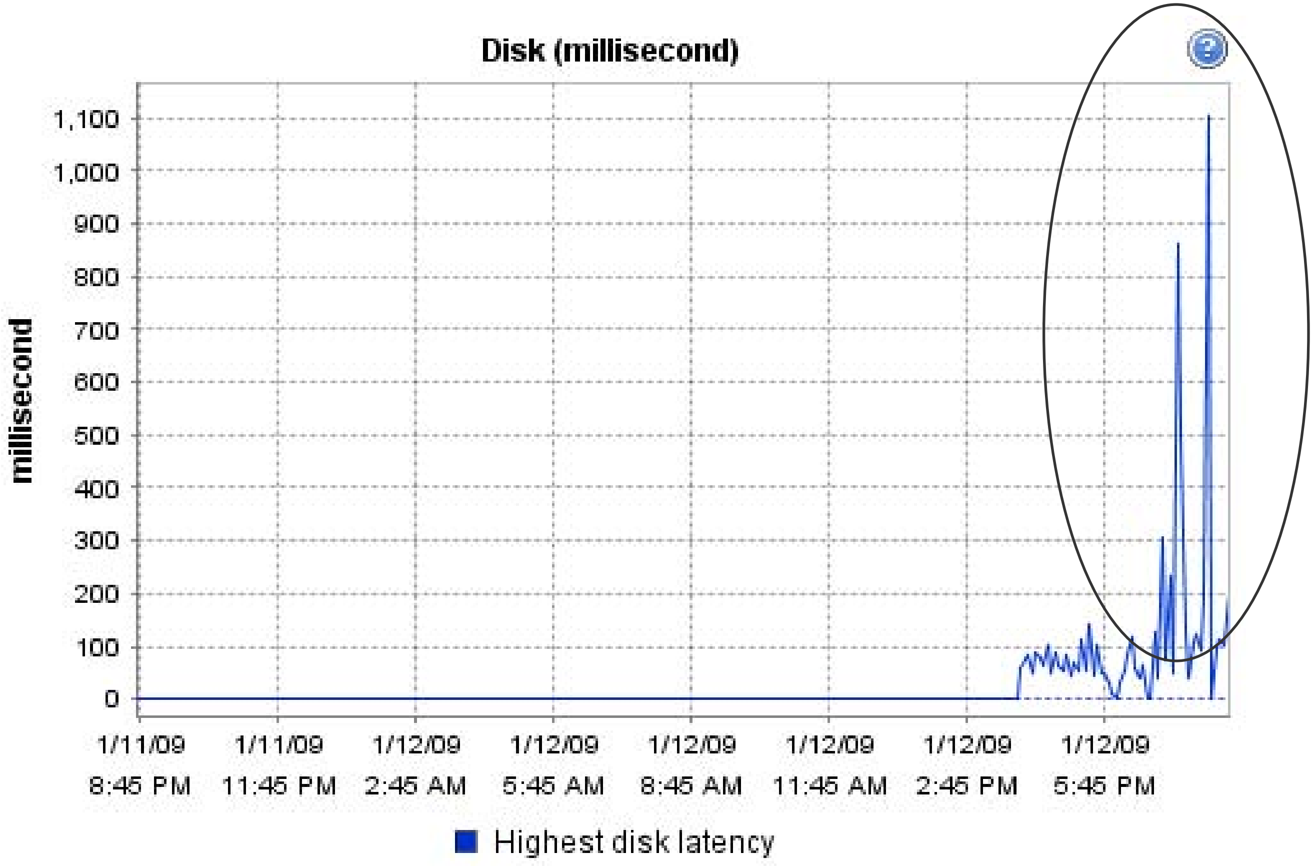
## Trying to Power on a VM

- Sometimes, powering on VM would take 5 seconds
- Other times, powering on VM would take 5 minutes!

## Where to begin?

- Powering on a VM requires disk activity on host ➔ Check disk metrics for host

# Examining Disk Latencies…




***Rule of thumb: latency > 20ms is Bad.***

Here:

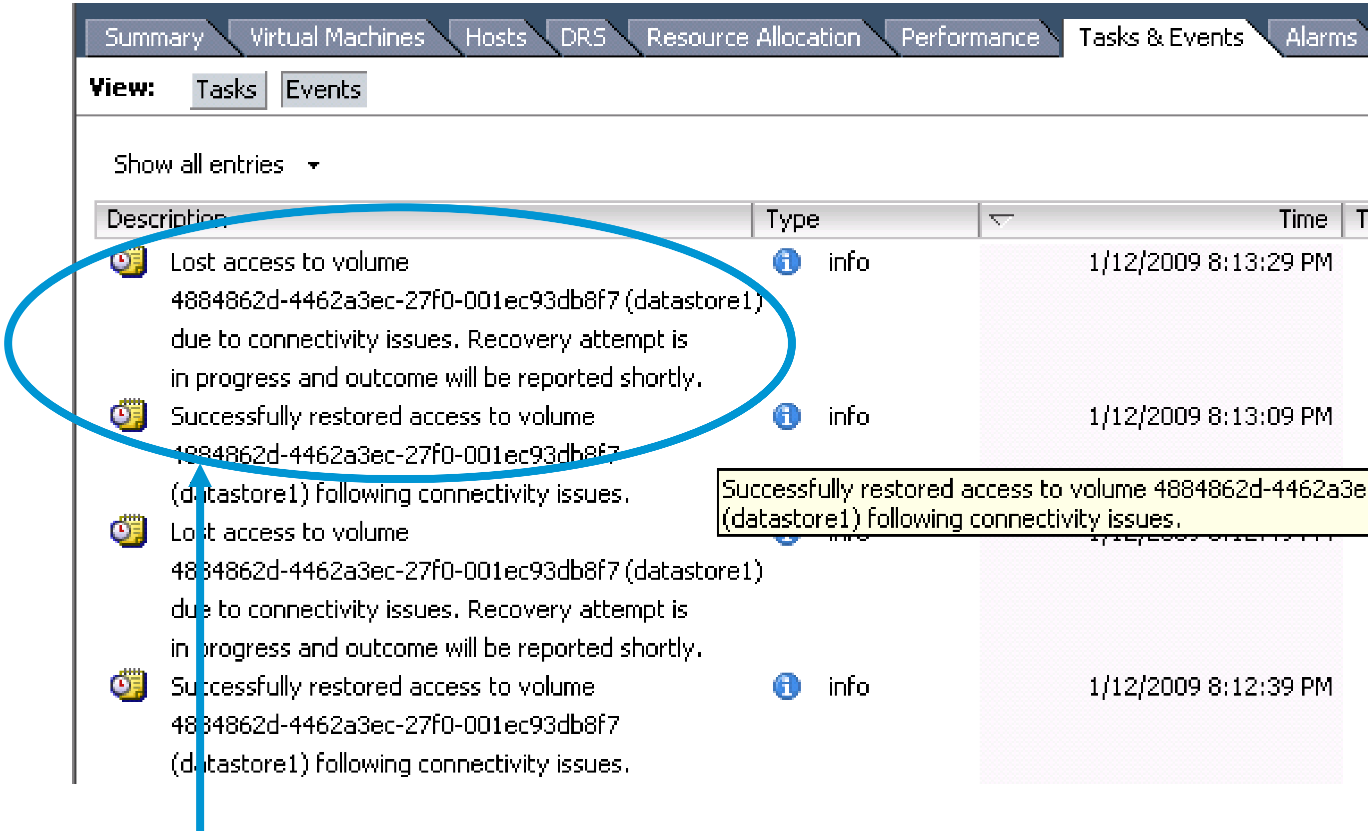
1,100ms

REALLY BAD!!!

→Chart shows highest disk latency for each 5-minute period

→Max Disk Latencies range from 100ms to 1100ms…very high! Why?

# High Disk Latency: Mystery Solved



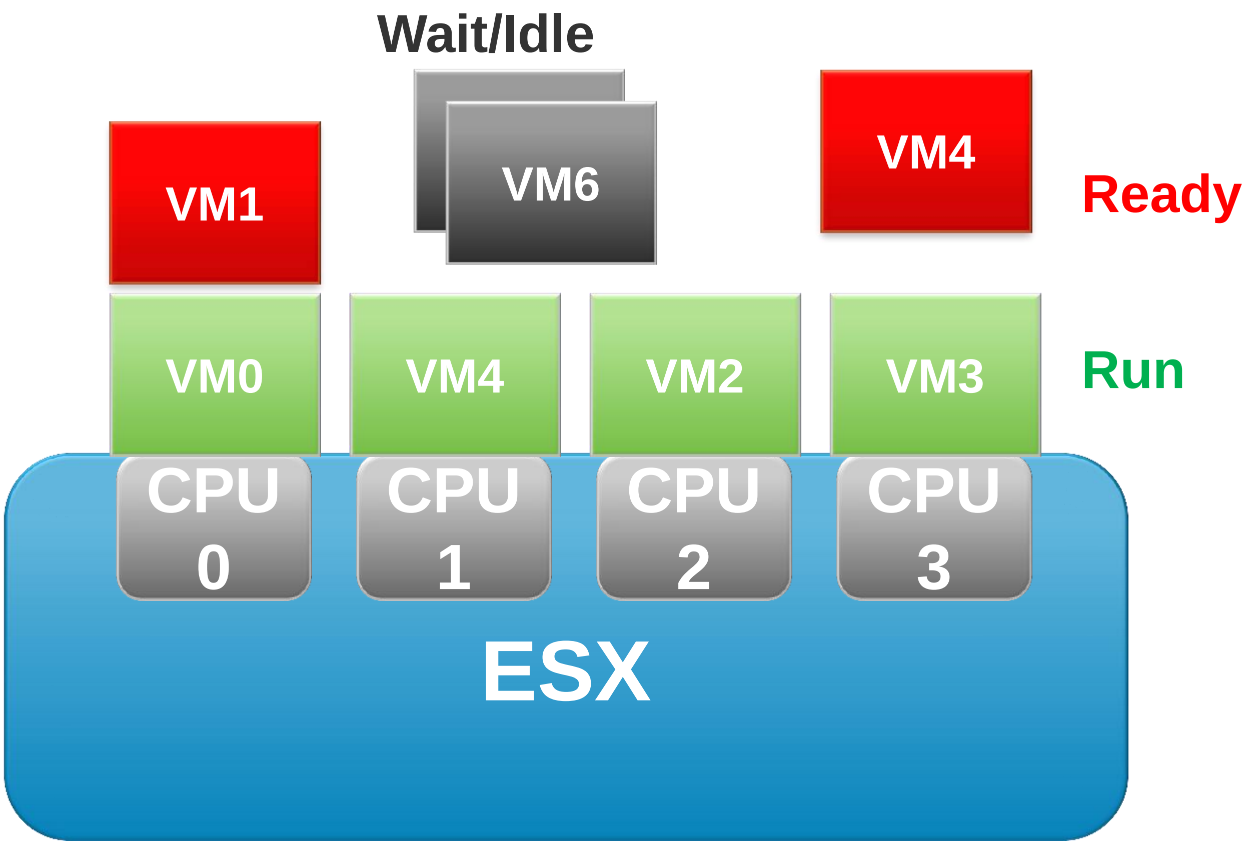

**Host events: disk has connectivity issues → high latencies!**

**Bottom line: correctness issue (bad disk controller) impacts performance!**



vmware

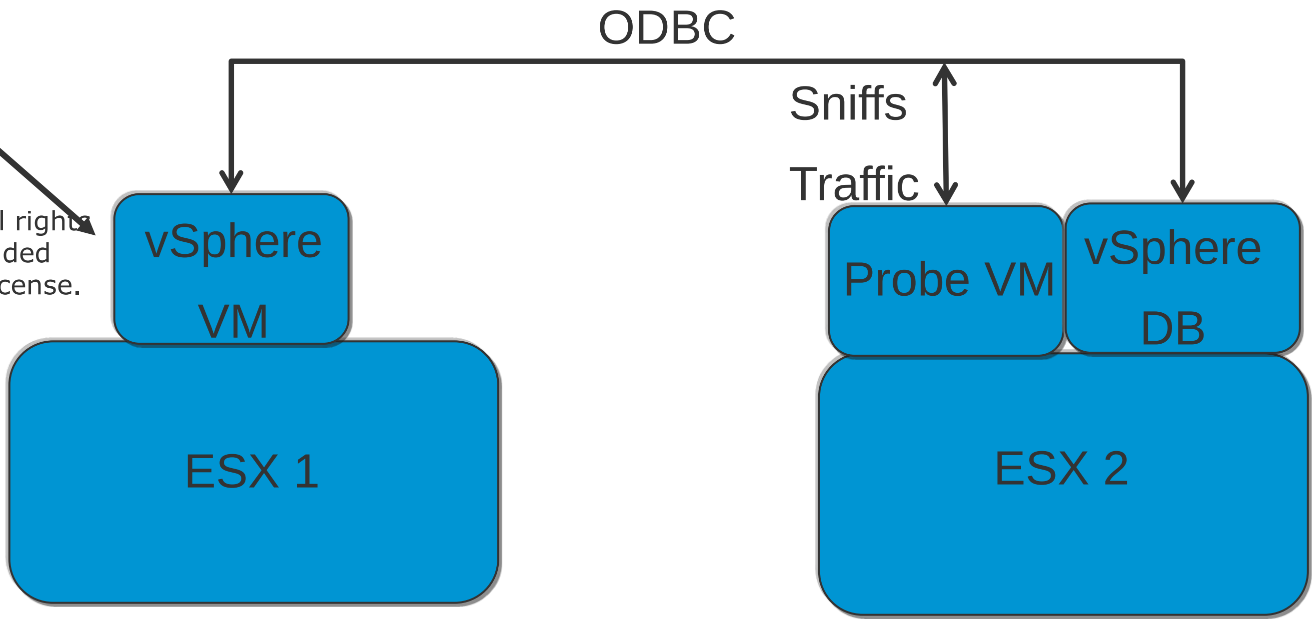
# Prelude to Case Studies 9 & 10: CPU Scheduling for VMs




Run (accumulating used time)
Ready (wants to run, no physical CPU available)
Wait: blocked on I/O or voluntarily descheduled

# Case Study 9: “But it’s only a small probe VM…”





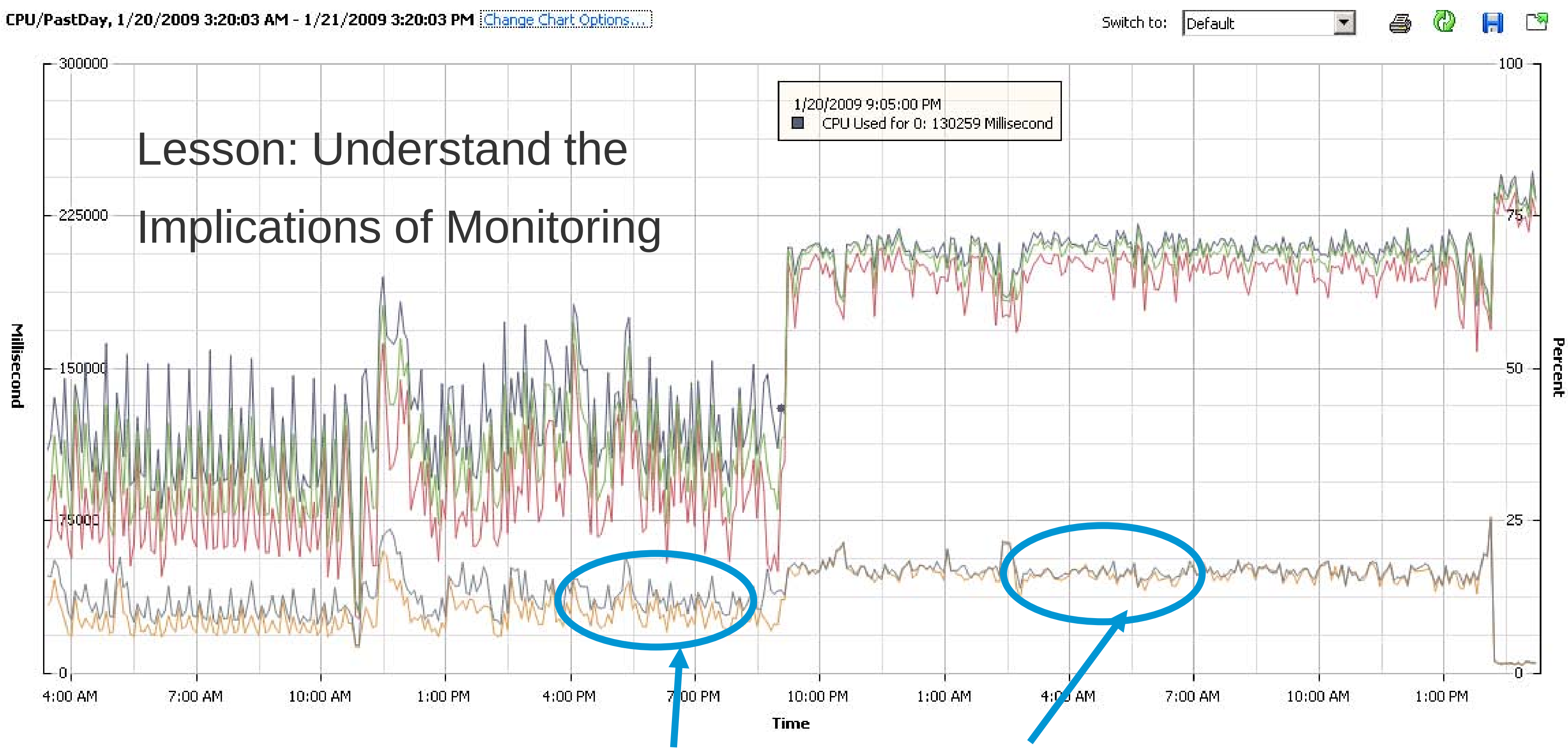

vSphere communicates with DB

Probe VM monitors vSphere-to-DB traffic

The more traffic, the more work done by Probe VM

User Complaint: vSphere VM is suddenly very unresponsive

# CPU Usage vs. Time for DB and Probe VM




DB VM ready time goes from 12.5% when idle to ~20% when user busy

DB ready time increases because Probe VM is busy

Probe VM takes CPU away from DB VM → user responsiveness suffers



vmware®

# Case Study #10: What Does This Metric Mean?

## Problem

- Customer Performs a Load Test: keeps attaching clients to a server
- At some point, CPU is NOT saturated, but latency starts to degrade
- At some point, client is unusable
- Why?

# “Oh yeah, it’s a disk problem…”

CPU Usage Increases…


Uh-oh! Disk Latencies go over a cliff!

# Hmm. Not So Fast!!!

**Problem:**

**Yes, Disk Latency gets worse at 4pm. (btw…due to swapping)**

**However, Application latency gets worse at 3:30pm!**

**What's going on from 3:30pm to 4pm?**

# Looking at a different chart…

| ID | GID | NAME | NWLD | %USED | %RUN | %SYS | %WAIT | %RDY | %IDLE | % |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | idle | 16 | 111.77 | 563.57 | 0.00 | 0.00 | 800.00 | 0.00 | |
| 2 | 2 | system | 7 | 0.01 | 0.02 | 0.00 | 700.00 | 0.00 | 0.00 | |
| 6 | 6 | helper | 73 | 0.15 | 0.25 | 0.00 | 7300.00 | 0.35 | 0.00 | |
| 7 | 7 | drivers | 9 | 0.00 | 0.01 | 0.00 | 900.00 | 0.01 | 0.00 | |
| 8 | 8 | vmotion | 4 | 0.00 | 0.00 | 0.00 | 400.00 | 0.00 | 0.00 | |
| 10 | 10 | console | 2 | 6.45 | 10.04 | 0.01 | 186.59 | 4.53 | 86.00 | |
| 15 | 15 | vmkapimod | 5 | 0.19 | 0.28 | 0.00 | 500.00 | 0.00 | 0.00 | |
| 17 | 17 | FT | 1 | 0.00 | 0.00 | 0.00 | 100.00 | 0.00 | 0.00 | |
| 18 | 18 | vobd.4279 | 8 | 0.00 | 0.00 | 0.00 | 800.00 | 0.00 | 0.00 | |
| 19 | 19 | net-cdp.4287 | 1 | 0.00 | 0.00 | 0.00 | 100.00 | 0.00 | 0.00 | |
| 20 | 20 | vmware-vmkauthd | 1 | 0.00 | 0.00 | 0.00 | 100.00 | 0.00 | 0.00 | |
| 68 | 68 | vm1 | 5 | 183.21 | 248.36 | 0.28 | 232.84 | 18.64 | 25.13 | |
| 69 | 69 | vm2 | 5 | 152.17 | 212.16 | 0.37 | 284.93 | 5.26 | 77.26 | |
| 70 | 70 | vm3 | 5 | 126.52 | 194.13 | 0.17 | 302.64 | 5.64 | 93.50 | |
| 71 | 71 | vm4 | 5 | 146.25 | 219.30 | 0.21 | 270.28 | 11.54 | 64.30 | |

%Used? %Run? What's the difference?

%used: normalized to base clock frequency

%run: normalized to clock frequency *while VM is running*…

%run > %used: Power Management is kicking in…

In this case, turn off power management→latency problems go away

*Lesson: understand your metrics!*



vmware

# The 10 Performance Issues I Mentioned

## 1. DB Lock % increase with decreasing load

- Be careful when you draw conclusions…

## 2. PowerCLI vs. Java

- Garbage-In, Garbage-Out: scalable solutions require careful design

## 3. Remote Console Issues

- Create APIs that are easy to use and difficult to abuse

## 4. 32-bit vs. 64-bit

- A small change can make a HUGE difference

## 5. “Logical” leak

- Just because you do “new/delete,” doesn’t mean memory won’t grow (btw., Java doesn’t save you!)
- Exaggerate a problem to make it easier to find the root cause

# The 10 Performance Issues I Mentioned

## 6. Slow memory growth until crash

- Sometimes you need customized profilers

## 7. Nagling

- Understand client/server interactions

## 8. Disk Latency

- Correctness Impacts Performance

## 9. Probe VM activity hurting performance of other VMs

- Understand the Impact of Monitoring

## 10. Power Management affecting Performance

- Understand your metrics & consider the whole system

# Conclusion: Tips for Performance Engineering

**Avoid assumptions! (see #10)**

**Understand the ENTIRE SYSTEM**

- Your code
- Other people's code
- Hardware

**Be persistent and thorough**

- Look at tons of metrics
- Look at behavior when things work as well as when they don't work