A Tale of 10 Bugs: Performance Engineering at VMware

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

ravi@vmware.com

MIT Guest Lecture, 6.172

12/13/11
Email thread from a colleague

“…

Interestingly, as the number of vcbench 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?
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
Case Studies in Performance Engineering @ VMware

Lessons Learned
Case Study #2: Garbage In, Garbage Out

Customer wants to draw this chart:

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

<table>
<thead>
<tr>
<th>Entities (cpu.usagemhz.average)</th>
<th>PowerCLI (Time in secs)</th>
<th>Java (Time in secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 VM</td>
<td>9.2</td>
<td>14</td>
</tr>
<tr>
<td>6 VMs</td>
<td>11</td>
<td>14.5</td>
</tr>
<tr>
<td>39 VMs</td>
<td>101</td>
<td>16</td>
</tr>
<tr>
<td>363 VMs</td>
<td>2580 (43 minutes)</td>
<td>50</td>
</tr>
</tbody>
</table>

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!*
PowerCLI

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

Java

Get VM ID
QueryAvailablePerfMetric();
QueryPerfCounter();
QueryPerfProviderSummary();
create PerfQuerySpec();
QueryPerf();
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 multiple client queries)
- Did not realize output format is verbose
- Did not realize # of RPC calls is $5 \times O(#\text{VMs})$

**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

User wants to view ‘console’ of a VM
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
CPU is mostly saturated (in 32-bit case, CPU is not saturated)
4(d) Look at Sampling Profile

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

From Root, most of the samples are from this call stack. Most popular stack, but is this the problem?
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

<table>
<thead>
<tr>
<th>Callers</th>
<th>Weight</th>
<th>% Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntdll.dll!ZwQueryVirtualMemory</td>
<td>3,123,003.752 ...</td>
<td>77.26</td>
</tr>
<tr>
<td>ntdll.dll! ?? ::FNODOBFM:: `string'</td>
<td>3,109,241.648 ...</td>
<td>76.92</td>
</tr>
<tr>
<td>ntdll.dll! MSVCR80.DLL!_RTDynamicCast</td>
<td>1,579,519.929 ...</td>
<td>39.07</td>
</tr>
<tr>
<td>ntdll.dll! MSVCR80.DLL!_RTtypeid</td>
<td>1,529,451.706 ...</td>
<td>37.84</td>
</tr>
<tr>
<td>ntdll.dll! [Root]</td>
<td>257.005 987</td>
<td>0.01</td>
</tr>
<tr>
<td>ntdll.dll! MSVCR80.DLL!CxxThrowException</td>
<td>13.006 609</td>
<td>0.00</td>
</tr>
<tr>
<td>ntdll.dll! [Root]</td>
<td>13,762.103 969</td>
<td>0.34</td>
</tr>
</tbody>
</table>

Not called a lot from root, however…

Called from few places and takes 77% CPU!

RTtypeid?
Hmm. RTtypeid is used in figuring out C++ type 39% of overall CPU? IncRef and DecRef are main callers
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](http://blogs.msdn.com/junfeng/archive/2006/10/17/dynamic-cast-is-slow-in-x64.aspx)
What Did We Learn?

RtTypeld is called from a bunch of places
RtTypeld is not, however, called from Root too often
RtTypeld 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 RtIToPcFileHeader)

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

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

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

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-imposed 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?
Host events: disk has connectivity issues → high latencies!
Bottom line: correctness issue (bad disk controller) impacts performance!
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

Lesson: Understand the Implications of Monitoring
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…

<table>
<thead>
<tr>
<th>ID</th>
<th>GID</th>
<th>NAME</th>
<th>NMLD</th>
<th>%USED</th>
<th>%RUN</th>
<th>%SYS</th>
<th>%WAIT</th>
<th>%RDY</th>
<th>%IDLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>idle</td>
<td>16</td>
<td>111.77</td>
<td>563.57</td>
<td>0.00</td>
<td>0.00</td>
<td>800.00</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>system</td>
<td>7</td>
<td>0.01</td>
<td>0.02</td>
<td>0.00</td>
<td>700.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>helper</td>
<td>73</td>
<td>0.15</td>
<td>0.25</td>
<td>0.00</td>
<td>7300.00</td>
<td>0.35</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>drivers</td>
<td>9</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>900.00</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>vmotion</td>
<td>4</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>400.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>console</td>
<td>2</td>
<td>6.45</td>
<td>10.04</td>
<td>0.01</td>
<td>186.59</td>
<td>4.53</td>
<td>86.00</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>vmkapiomod</td>
<td>5</td>
<td>0.19</td>
<td>0.28</td>
<td>0.00</td>
<td>500.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>FT</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>vobd.4279</td>
<td>8</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>800.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>net-cdp.4287</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>vmware-vmkauthd</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>68</td>
<td>68</td>
<td>2087STUTV1V0601</td>
<td>5</td>
<td>183.21</td>
<td>248.36</td>
<td>0.28</td>
<td>232.84</td>
<td>18.64</td>
<td>25.13</td>
</tr>
<tr>
<td>69</td>
<td>69</td>
<td>2087STUTV1V0604</td>
<td>5</td>
<td>152.17</td>
<td>212.16</td>
<td>0.37</td>
<td>284.93</td>
<td>5.26</td>
<td>77.26</td>
</tr>
<tr>
<td>70</td>
<td>70</td>
<td>2087STUTV1V0603</td>
<td>5</td>
<td>126.52</td>
<td>194.13</td>
<td>0.17</td>
<td>302.64</td>
<td>5.64</td>
<td>93.50</td>
</tr>
<tr>
<td>71</td>
<td>71</td>
<td>2087STUTV1V0602</td>
<td>5</td>
<td>146.25</td>
<td>219.30</td>
<td>0.21</td>
<td>270.28</td>
<td>11.54</td>
<td>64.30</td>
</tr>
</tbody>
</table>

%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!
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
Bonus…bugs that didn’t make the cut (but are still cool 😊)

Admin wants to clone a VM

1. User talks to management server
2. Management server locates VM on host
3. Host copies VM to another host
Bonus…bugs that didn’t make the cut (but are still cool 😊)
Complaint: I upgraded software, and now clone is slower!

Lessons:
1. Try to change 1 thing at a time
2. Try to do apples-to-apples comparison!

1. Customer changed management layer:
   → Clone slowed down
2. However, customer ALSO changed host version
   → New (unqualified) device driver was culprit!
   → Clones to old hosts were actually the same speed
One more random lesson

Sometimes, it’s important to not try too hard

• Anomaly detection shows red
• Investigation shows that memory usage is anomalous
• Why? We were in a hands-on-lab: memory spiked when people entered. Duh!
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