Anecdotes from a Performance Engineer

Lessons learned from 19 years of Performance Engineering

Ravi Soundararajan, Performance Team, VMware
Guest Lecture: Engineering Robust Server Software
April 6, 2023
The Key to Good Performance

“Make the common case fast…

…but make sure it is correct…

…and make sure uncommon cases are correct, too…”

(By the way, make sure it really is the common case)
Making the Common Case Fast: VMware

Example: VMware Fusion (running Windows on a Mac) or VMware Workstation

Common case: User-level code running directly on HW (no emulation)
“What else does VMware do?”
What *else* does VMware do? Server Consolidation

Virtualization takes this…

300 Servers
...and Replaces It With This

300 Servers without VMware software

8 Servers, 1 rack with VMware software
vMotion (Thanks, Sree)
vMotion

1. Pre-copy Memory (over network)

2. StunVM

3. Copy Device State + Memory

4. Switch VM Disk Pointer

Requirement: < 1s stun time. Change-block tracking critical
Storage vMotion...can we use the same logic?

1. Copy VM home to new location
2. Start changed block tracking
3. Pre-copy disk to destination (multiple iterations)
4. Copy VM home to new location
5. Delete original VM home and disks

Fast suspend, copy all remaining disk regions, resume VM
Transfer to Dirty Ratio (TDR)

Transfer to Dirty Ratio (model)

Duration of each update operation

Precopy Iterations

success

failure

"1/2"
"2/3"
"3/4"
"4/5"
"5/6"
"6/7"
"7/8"
"8/9"
"9/10"
"34/36"
"1/36"
Tracking modifications...

Host copies src to dst in blocks

SRC mod? Mirror writes

SRC mod: OK, will get later

Source

Destination
Stories from my job
A Database Outage
Rollup caused DoS to DB...Application crashed

Internal Testing: 6 days, 23 hours, 56 minutes. Customer outage at 7 days
Lesson: Do long-pole testing and test ‘inflection’ points
The right tool for the job...
The Symptom: Concurrent Add Host Regression

- Colleague saw new logs: “Remediate ... providers...”

Baseline: 32s
Regressed: 37s
Simple Log Analysis: Find the Biggest Gap

We are spending time in remediate, but how do we link to the regression?

Baseline: 19s

Regression: 26s

How to link to add host?

CPU for regression case was not saturated? Locks?
Cutting to the chase (1/2)

“Obviously, code indicates we are doing more work in the regressed case.”

Me: You are doing more in this code path.

Dev team: But it is in a separate thread...

Me: Ugh. Ok, let me prove my point.
Changes in other threads…”should not slow down add host”

CacheUpdate occurs in both runs (scheduled IMMEDIATELY)
ScheduleRemediation is a no-op in the baseline case
(and scheduled IMMEDIATELY in regressed case)
Are we CPU saturated?
Problem: granularity is too coarse

Usage hovers around 75%...not CPU saturated
Using a CPU Profiler (N.B., Zoom, End-of-Life product)

"After" spends more time in locks
Debugging: Lock profiling

Slower EXCLUSIVEALL locks

Baseline

<table>
<thead>
<tr>
<th>Time</th>
<th>Message</th>
<th>File</th>
<th>Line</th>
<th>Severity</th>
<th>Description</th>
</tr>
</thead>
</table>

Regression

<table>
<thead>
<tr>
<th>Time</th>
<th>Message</th>
<th>File</th>
<th>Line</th>
<th>Severity</th>
<th>Description</th>
</tr>
</thead>
</table>

© 2019 VMware, Inc.
Great Performance Books

Buy these!

**Systems Performance: Enterprise and the Cloud, 2nd Edition (2020)**

This is the official site for the book Systems Performance: Enterprise and the Cloud, 2nd Edition, published by Addison-Wesley (2020). Here I’ll describe the book, link to related content, and list errata.

You can find it on Amazon as paperback and Kindle, and on InformIT as paperback, EPUB, MOBI, and PDF. (If you purchase through the Amazon or InformIT links, the book's technical editor earns a commission.)

The first edition has been very successful, becoming recommended or required reading at many companies, and has sold editions translated to Chinese, Japanese, Polish, and Korean. I’ve had many emails from people studying the book for the Facebook engineering interview. I’m glad it’s helpful.

There is also a companion book, BPF Performance Tools, that provides advanced coverage of BPF performance analysis tools.


**BPF Performance Tools (book)**

This is the official site for the book BPF Performance Tools: Linux System and Application Observability, published by Addison Wesley (2019). This book can help you get the most out of your systems and applications, helping you improve performance, reduce costs, and solve software issues. Here I’ll describe the book, link to related content, and list errata and updates.

The book is available on Amazon.com (paperback, kindle), InformIT (paperback, PDF, etc), and Safari (here and here). The paper book was released in December 2019 but sold out immediately. ISBN-13: 9780136554820. (If you purchase through the Amazon or InformIT links, the book's technical editor earns a commission.)

The Amazon Kindle preview shows the first 100 pages out of this 880 page book.

There is also a companion book, Systems Performance: 2nd Edition (2020), that provides advanced coverage of performance analysis.

BCC Tools

Helpful tools built on top of BPF
BCC: Runqlat. How much time is spent waiting to run?
runqlat -P 160 > /storage/core/runqlat-60.txt &

Before

<table>
<thead>
<tr>
<th>pid = 36081</th>
<th>usescs</th>
<th>count</th>
<th>distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 -&gt; 1</td>
<td>21103</td>
<td>**********************</td>
</tr>
<tr>
<td></td>
<td>2 -&gt; 3</td>
<td>18468</td>
<td>************************</td>
</tr>
<tr>
<td></td>
<td>4 -&gt; 7</td>
<td>3528</td>
<td>************************</td>
</tr>
<tr>
<td></td>
<td>8 -&gt; 15</td>
<td>3652</td>
<td>*****</td>
</tr>
<tr>
<td></td>
<td>16 -&gt; 31</td>
<td>3558</td>
<td>*****</td>
</tr>
<tr>
<td></td>
<td>32 -&gt; 63</td>
<td>3437</td>
<td>*****</td>
</tr>
<tr>
<td></td>
<td>64 -&gt; 127</td>
<td>3965</td>
<td>*****</td>
</tr>
<tr>
<td></td>
<td>128 -&gt; 256</td>
<td>2173</td>
<td>****</td>
</tr>
<tr>
<td></td>
<td>256 -&gt; 511</td>
<td>1081</td>
<td>**</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>pid = 35069</th>
<th>usescs</th>
<th>count</th>
<th>distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 -&gt; 1</td>
<td>17161</td>
<td>**********************</td>
</tr>
<tr>
<td></td>
<td>2 -&gt; 3</td>
<td>20018</td>
<td>************************</td>
</tr>
<tr>
<td></td>
<td>4 -&gt; 7</td>
<td>3232</td>
<td>************************</td>
</tr>
<tr>
<td></td>
<td>8 -&gt; 15</td>
<td>3933</td>
<td>*****</td>
</tr>
<tr>
<td></td>
<td>16 -&gt; 31</td>
<td>3919</td>
<td>*****</td>
</tr>
<tr>
<td></td>
<td>32 -&gt; 63</td>
<td>3446</td>
<td>*****</td>
</tr>
<tr>
<td></td>
<td>64 -&gt; 127</td>
<td>2518</td>
<td>****</td>
</tr>
<tr>
<td></td>
<td>128 -&gt; 256</td>
<td>1279</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td>256 -&gt; 511</td>
<td>589</td>
<td>*</td>
</tr>
</tbody>
</table>

Longer, but doesn’t explain everything...
### Before (example, one time step)

<table>
<thead>
<tr>
<th>runqocc</th>
<th>CPU</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>runqocc</td>
<td>0</td>
<td>3.03%</td>
</tr>
<tr>
<td>runqocc</td>
<td>1</td>
<td>10.10%</td>
</tr>
<tr>
<td>runqocc</td>
<td>2</td>
<td>14.14%</td>
</tr>
<tr>
<td>runqocc</td>
<td>3</td>
<td>5.05%</td>
</tr>
<tr>
<td>runqocc</td>
<td>4</td>
<td>3.03%</td>
</tr>
<tr>
<td>runqocc</td>
<td>5</td>
<td>4.04%</td>
</tr>
<tr>
<td>runqocc</td>
<td>6</td>
<td>10.10%</td>
</tr>
<tr>
<td>runqocc</td>
<td>7</td>
<td>4.04%</td>
</tr>
<tr>
<td>runqocc</td>
<td>8</td>
<td>5.05%</td>
</tr>
<tr>
<td>runqocc</td>
<td>9</td>
<td>13.13%</td>
</tr>
<tr>
<td>runqocc</td>
<td>10</td>
<td>4.04%</td>
</tr>
<tr>
<td>runqocc</td>
<td>11</td>
<td>5.05%</td>
</tr>
<tr>
<td>runqocc</td>
<td>12</td>
<td>13.13%</td>
</tr>
<tr>
<td>runqocc</td>
<td>13</td>
<td>4.04%</td>
</tr>
<tr>
<td>runqocc</td>
<td>14</td>
<td>5.05%</td>
</tr>
<tr>
<td>runqocc</td>
<td>15</td>
<td>5.05%</td>
</tr>
</tbody>
</table>

### After (example, one time step)

<table>
<thead>
<tr>
<th>runqocc</th>
<th>CPU</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>runqocc</td>
<td>0</td>
<td>10.10%</td>
</tr>
<tr>
<td>runqocc</td>
<td>1</td>
<td>11.11%</td>
</tr>
<tr>
<td>runqocc</td>
<td>2</td>
<td>3.03%</td>
</tr>
<tr>
<td>runqocc</td>
<td>3</td>
<td>4.04%</td>
</tr>
<tr>
<td>runqocc</td>
<td>4</td>
<td>9.09%</td>
</tr>
<tr>
<td>runqocc</td>
<td>5</td>
<td>8.08%</td>
</tr>
<tr>
<td>runqocc</td>
<td>6</td>
<td>10.10%</td>
</tr>
<tr>
<td>runqocc</td>
<td>7</td>
<td>5.05%</td>
</tr>
<tr>
<td>runqocc</td>
<td>8</td>
<td>7.07%</td>
</tr>
<tr>
<td>runqocc</td>
<td>9</td>
<td>14.14%</td>
</tr>
<tr>
<td>runqocc</td>
<td>10</td>
<td>9.09%</td>
</tr>
<tr>
<td>runqocc</td>
<td>11</td>
<td>2.02%</td>
</tr>
<tr>
<td>runqocc</td>
<td>12</td>
<td>11.11%</td>
</tr>
<tr>
<td>runqocc</td>
<td>13</td>
<td>12.12%</td>
</tr>
<tr>
<td>runqocc</td>
<td>14</td>
<td>5.05%</td>
</tr>
<tr>
<td>runqocc</td>
<td>15</td>
<td>10.10%</td>
</tr>
</tbody>
</table>
Runqlen before vs. after (Day 1)
Graphically...using number of items in queue (runqlen 1 60)

Runq looks longer for regression...but not conclusive (see next slide)
Runqlen before vs. after (Day 2)

Runq looks longer for baseline in this case...
Flamescope: CPU activity sub-second heat maps

Flamescope: cpu activity at sub-second granularity

https://github.com/Netflix/flamescope/archive/refs/heads/master.zip

Before

Activity stops around 37s
Flamescope heatmap of CPU activity

More activity than baseline after 37s
(Looks promising…there is more, but that is for another time)
Flamescope heatmap of CPU activity

Before

After

More activity than baseline after 37s
(Looks promising…there is more, but that is for another time)
BPF script for tracking locks and counting add host calls (1/3)

Initialize like AWK

Uprobe: user-level probe at entry of MoLockIermpl::MoLockerImpl

Initialize array entry

Count() : counts occurrences

```c
#!/usr/share/bpftrace
/*
 * lockaddhostint.bt - Count number of MoLock calls and latency of addHostInt
 * 08-Sep-2021  Ravi Soundararajan  Created this.
 */

BEGIN
{
    printf("Tracing MoLocker and addHostInt calls, Ctrl-C to end.\n");
    @addhostcount = 0;
    // Trace locks only when you start seeing addHost
    @addhoststarted = 0;
}

uprobe:/usr/lib/vmware-vpx/vpxd:MoLockerImpl*MoLockerImpl
($1 == 0 || pid == $1) && (@addhoststarted == 1)/
{
    @lock_start[tid] = nsecs;
    // Depending on the call, the second arg is either a lock mode or pointer
    // (10 is just a guess for now)
    if (arg1 < 10 ) {
        @lock_mode[arg1] = count();
    } else {
        // The 'lock mode' argument is a pointer.
        @lock_mode[*arg1] = count();
    }
}
BPF script for tracking locks and add host calls (2/3)

nsecs: current time

Do stuff on function exit

Hist: create histogram
Bpf script for tracking locks and add host calls (3/3)

1. Compute latency
2. Store in histogram
3. Cleanup
Sample output (with symbols resolved)

```bash
root@localhost [ /storage/core/bpf-perf-tools-book ]# bptrace lockaddhostint-scope.bt 'pidof vpxd' 64
Attaching 48 probes...
Tracing MoLocker and addHostInt calls, Ctrl-C to end.

@addhost_latency_ms:
[4K, 8K)  13 |aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa|
[8K, 16K) 51 |bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb|

@addhostcount: 64
@addhoststarted: 1

@lock_latency_ms[Snapshot::Cluster:aaaaaaa, std::allocator<Snapshot::Cluster>]: 1 |cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
What did I see?

Only in the regressed run: locks in KClient

```plaintext
@lock_latency_ms[VMcrypt::CryptoManagerImpl::Remediate(HostMo*, std::list<VMcrypt::]
(0] 70 |-----------------------------------------------|
[1] 17 |-----------------------------------------------|
[2, 4] 22 |-----------------------------------------------|
[4, 8] 17 |-----------------------------------------------|
[8, 16] 0 |                                |
[16, 32] 2 |@                                |

@lock_latency_ms[VMcrypt::IsInCluster(HostMo*) + 76]:
[0] 64 |-----------------------------------------------|

@lock_latency_ms[VMcrypt::KClient::KClient(HostMo*, Vmacore::Service::Logger*) + 378]:
[0] 1266 |-----------------------------------------------|
[1] 6 |                                |
[2, 4] 6 |                                |
[4, 8] 5 |                                |

@lock_latency_ms[Vpxd::GetHostSslThumbprint(HostMo*) + 97]:
[0] 1271 |-----------------------------------------------|
[1] 6 |                                |
[2, 4] 3 |                                |
[4, 8] 1 |                                |
[8, 16] 2 |                                |
```
Understanding your application
Hierarchical Locking

Parent

Child 1
- Leaf 1
- Leaf 2

Child 2
- Leaf 3

EXCLUSIVEALL or SHAREALL
BCC, BPF, and Logs

Logs

Precise but application-dependent

BCC/BPF

Visibility without application recompilation

More than just CPU
What about Memory?
Chap Memory Analysis Tool (thanks, Tim Boddy!)
https://github.com/vmware/chap

Analyzes core files
• Leaks
• Memory growth
• Corruption

Support
• C++ (full)
• Java/Python/Go (limited)
Chap in a nutshell

Given a core dump...

Figure out data types

Figure out what points to what

Figure out memory allocation sizes

Provide commands to browse the above

Great complement to gdb
# Count the set formed by starting with all Orange allocations then extending to # any allocations reachable by traversing one or more outgoing references.

count allocation Orange /extend ->
Sample output

**chap> summarize used**

Unsigned allocations have **20742270 instances** taking **0x3a16dbdc8 (15,593,225,672) bytes.**

Unsigned allocations of size 0x48 have **84331647 instances** taking **0x169e983b8 (6,071,878,584) bytes.**

Unsigned allocations of size 0x18 have **45382444 instances** taking **0x40eb8c20 (1,089,178,656) bytes.**

... 

Signature **55e8cd940848** (Vmomi::Array<long>) has **66771769 instances** taking **0xa7ccdd58 (2,815,221,080) bytes.**

Signature **55e8cd9ac860** (VimUtil::View::NotifyEvent) has **5766819 instances** taking **0x134da5a8 (323,855,784) bytes.**

Signature **55e8cd93ee78** (Vmomi::Array<std::string>) has **2517883 instances** taking **0x6124678 (101,860,984) bytes.**

...
Sample output

Lots of free memory? Is fragmentation a problem?

chap> summarize free

Unsigned allocations have **53892814 instances** taking 0x34d7f7968(14,185,101,672) bytes.

Unsigned allocations of size 0x78 have **10269381 instances** taking 0x4973cc58(1,232,325,720) bytes.

Unsigned allocations of size 0x48 have **6591974 instances** taking 0x1c4a28b0(474,622,128) bytes.

...

chap> enumerate used /size 48

55e8cea8aee0
55e8cea8af30
55e8cea8af80
55e8cea8b3d0

...
Anchored allocation at 55e8cea8b3d0 of size 48

This allocation matches pattern COWStringBody.

This has capacity 34, reference count 1 and a string of size 34 containing "d4d63a7d-0a4547858b3b-0b9d1f27bb01".

# Allocation at 0x55e8cea8b3d0 is referenced by allocation at 0x7fa452c0c6d0.

# Allocation at 0x7fa452c0c6d0 will be extended in state StopHere.

Anchored allocation at 7fa452c0c6d0 of size 3e48

This allocation matches pattern VectorBody.

Only the first 0x3e40 bytes are considered live.
chap> summarize used

Signature 556e94b5f850 (Vmomi::Array<long>) has 63697185 instances taking 0x9b3f49d8 (2,604,616,152) bytes.

Signature 7ff1e1219270 (Vim::PerformanceManager::IntSeries) has 63688349 instances taking 0x990ece58 (2,567,884,376) bytes.

Signature 7ff1e1219130 (Vim::PerformanceManager::MetricId) has 63684368 instances taking 0x642b17c0 (1,680,545,728) bytes.

DB could not keep up with stats writes: caused queuing in App Server

- Needed to compact in-memory stats, do native sql batching to speed up database write

Lessons:

1. Understand the queues in your system!
2. Do Long-pole tests
CPU Profiling: Linux Perf

(because Zoom is EOL)
Top utility: chap takes 100% CPU
Linux Perf

Install linux-tools-`uname -r` and do `echo 0 > /proc/sys/kernel/kptr_restrict`

```
perf record -p `pidof vpxd` --call-graph dwarf -F 99 -o vpxd-60s-clone.data sleep 60

perf report -i vpxd-60s-clone.data -g --sort comm,dso
```
Perf utility showing Chap CPU usage

Top-down overall view

```
% perf record -call-graph dwarf-F 99 -p `pidof my-chap` -- sleep 60
% perf report -i perf.data -g
```
Perf utility running against chap: 2\textsuperscript{nd} level view
Interesting bugs or observations
Cloning a VM

Diagram:
- S1
- D1
- Network
- local
- v1
- SAN
Intro: Cloning a VM

Actual slide from 2004

**Provisioning Operations: Cloning (2)**

<table>
<thead>
<tr>
<th>Src Disk</th>
<th>Dst Disk</th>
<th>1GB VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Local</td>
<td>2m 31s</td>
</tr>
<tr>
<td>Local</td>
<td>SAN</td>
<td>2m 32s</td>
</tr>
<tr>
<td>SAN</td>
<td>Local</td>
<td>1m 27s</td>
</tr>
<tr>
<td>SAN</td>
<td>SAN</td>
<td>1m 32s</td>
</tr>
<tr>
<td>SAN</td>
<td>SAN_2</td>
<td>1m 30s</td>
</tr>
</tbody>
</table>

- Local -> SAN slower than SAN -> Local
Cloning Overview: SAN to Local (Fast)

Destination-initiated
Destination has access to local and SAN datastore
Cloning Overview: Local to SAN (Slow)

Destination-initiated
Destination does not have access to source and destination datastore
Tuning to a platform

OSes
Switching OSes

Performance over time (higher is better)
Photon Performance
Many changes...which one is required?

Photon8 has numerous changes:
• Idle tickless mode
• NUMA BALANCING disabled
• 250Hz timer
• Stack protector disabled

Which is required?
Photon Performance
Many changes...which one is required?

Photon8 default config has ALL changes

<table>
<thead>
<tr>
<th>Configuration</th>
<th>SLES11 ops/min</th>
<th>SLES12 ops/min</th>
<th>Photon8 ops/min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default config</td>
<td>5600 ops/min</td>
<td>4650 ops/min</td>
<td>6111 ops/min (all changes)</td>
</tr>
</tbody>
</table>

- NUMA_BALANCING caused performance drop
- NUMA_BALANCING new in kernel 3.8.0
- SLES11: 3.0; SLES12: 3.12; Photon: 4.2.0
Photon Performance
Many changes...which one is required?

Photon8 default config has ALL changes

<table>
<thead>
<tr>
<th>Configuration</th>
<th>SLES11 ops/min</th>
<th>SLES12 ops/min</th>
<th>Photon8 ops/min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default config</td>
<td>5600 ops/min</td>
<td>4650 ops/min</td>
<td>6111 ops/min (all changes)</td>
</tr>
<tr>
<td>NUMA_BALANCING ENABLED</td>
<td>n/a</td>
<td>4861 ops/min</td>
<td>4558 ops/min</td>
</tr>
</tbody>
</table>

- NUMA_BALANCING caused performance drop
- NUMA_BALANCING new in kernel 3.8.0
- SLES11: 3.0; SLES12: 3.12; Photon: 4.2.0
Photon Performance
Many changes...which one is required?

Photon8 default config has ALL changes

<table>
<thead>
<tr>
<th>Configuration</th>
<th>SLES11 ops/min</th>
<th>SLES12 ops/min</th>
<th>Photon8 ops/min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default config</td>
<td>5600 ops/min</td>
<td>4650 ops/min</td>
<td>6111 ops/min (all changes)</td>
</tr>
<tr>
<td>NUMA_BALANCING ENABLED</td>
<td>n/a</td>
<td>4861 ops/min</td>
<td>4558 ops/min</td>
</tr>
<tr>
<td>NUMA_BALANCING DISABLED</td>
<td>n/a</td>
<td>6023 ops/min</td>
<td>6108 ops/min</td>
</tr>
</tbody>
</table>

- NUMA_BALANCING caused performance drop
- NUMA_BALANCING new in kernel 3.8.0
- SLES11: 3.0; SLES12: 3.12; Photon: 4.2.0
The Importance of a Model

Host Syncs and Scaling
Host Sync

RegisterVM Benchmark

Latency of registering 300th VM?

Shape of curve? Linear? Flat?
Host Syncs

Chart SP_Reg.4: Concurrent VM Registration On Single Host Using ViM To VC
(10 Concurrent Registrations)

- Raw's crossported upx + hostd + alarmflog off
- 2.0.1 GA vpxa
- Raw's crossported upx + hostd - alarmflog on
- Trendline (Raw's crossported upx + hostd + alarmflog off)
- Trendline (2.0.1. GA vpxa)
- Trendline (Raw's crossported upx + hostd - alarmflog off)

Update 1: Quadratic, not flat

Update 1 + fix: Closer to linear/flat
Understanding Scaling

• RegisterVM quadratic... Why?

• foundHandle = GetExistingHandleForPathName(...)

• Check over ALL existing handles for duplicate handle ➔ scales with # of VMs

• “Ravi, when we deferred it, we didn't consider the impact of this performance improvement for XXX. We will retriage this and get back to you shortly.”
APIs

Toolkits and APIs: pitfalls and perils
Default PowerCLI  (https://communities.vmware.com/thread/499845)

```powershell
{ForEach ($esx in Get-VMHost) {
    $vCPU = Get-VM -Location $esx | Measure-Object -Property NumCpu -Sum | select -ExpandProperty Sum

    ...
}

~20 hosts, 300 VMs: 80s

Client-side filtering: server sends all data, and client filters…
PowerCLI (2 of 2)

Faster PowerCLI: Pull Get-VM and Get-VMHost out of loop

```powershell
$allVMs = Get-VM
$allHosts = Get-VMHost

&(ForEach ($esx in $allHosts) {
    $vCPU = $allVMs | where {$_.vmhost.name -eq $esx.name} | Measure-Object ...
})
```

~20 hosts, 300 VMs: 7.5s (10x speedup)

> Code samples != production-quality, scalable code.

> Need better guidance for consumers
ContainerView Leaks

Question: Memory of virtual appliance growing. Why?

Look at memory growth
def print_vm_host(self):
    container = self.service_content.rootFolder  # where to start
    viewType = [vim.HostSystem, vim.VirtualMachine]  # entities you care about
    recursive = True

    containerView = self.service_content.viewManager.CreateContainerView(container, viewType, recursive)  # create view

    children = containerView.view
    for child in children:
        print child.name

    containerView.DestroyView()  # destroy view
ContainerView Creation/Deletion

Create >> Destroy

- ContainerView Leak
- Bug in extension (now fixed)

https://kb.vmware.com/s/article/2151324

Try to make platform resilient
Improve API docs and usability
Visualization
Grafana, Prometheus, OpenTracing, Wavefront, etc.
Back to Basics

Java and C++
Java memory management is done by the Java virtual machine. Garbage Collection: Find ‘unreachable objects’ and delete them.

Java Garbage Collection
“Mark, sweep, and compact” garbage collector:
• Mark: identify garbage
• Sweep: find garbage on heap, de-allocate it
• Compact: collect all live memory together

Java Memory (not including code cache)
Java GC and Tuning Notes

GC for Eden is frequent and hopefully low overhead

GC for “Oldgen” is less frequent and more CPU-intensive than Eden

Rough guideline: most (80%?) of memory is short-lived

Many tunables in Java:
• Heap sizes (-Xms, -Xmx)
• Desirable ‘free heap’ ratio
• Survivor-to-Eden ratio
• Type of GC (serial, concurrent, mark/sweep, etc.)
• Number of GC threads
• Stack size (thread stacks NOT part of heap memory)
• Permgen size (not part of heap)

Profiling tools
• Yourkit, VisualVM, JMX counters, etc.
Pathological Memory Usage for a Java Process

Min heap = max heap ➔ Less incentive to GC

Eden
(Survivor negligible)

OldGen Growth: 5GB to 20GB!
Few OldGen GCs.
Why?

Min = max? Usually good only if you know what you need.
Fixing the pathology

--Shrink max heap setting

--Do not set Xms (initial heap). Do not set initial Permgen
Question: How efficient is your software?
VMware software spans many layers:

• Virtual Machine monitor
  – Needs small footprint for best performance
  – Any CPU cost becomes virtualization overhead: slower guests

• Kernel

• Higher-level application software

➤ Efficiency matters everywhere
RDTSC: read timestamp counter
Lets you see the number of cycles for a section of code

```c
#if defined(__x86_64__)

static __inline__ unsigned long long rdtsc(void)
{
    unsigned hi, lo;

    __asm__ __volatile__("rdtsc" : "=a"(lo), "=d"(hi));

    return ( (unsigned long long)lo) | ( ((unsigned long long)hi)<<32 );
}
#endif
```
Rdtsc string vs. integer compare

```c
int t1 = rdtsc();

for (int i = 0; i < num_iters; i++) {
    int equal = strncmp(s1, s2, strlen(s1));
}

int t2 = rdtsc();
```

String comparison:
81 cycles per loop

```c
int t1 = rdtsc();

for (int i = 0; i < num_iters; i++) {
    int equal = (num1 == num2);
}

int t2 = rdtsc();
```

Integer comparison:
6 cycles per loop
Strncmp: 81 cycles

% objdump -S -l -C test

equal = strncmp(s1, s2, strlen(s1));

400aa7:       48 8b 45 f0 mov     0xffffffffffffff0(%rbp),%rax
400aab:       48 c7 c1 ff ff ff ff mov     $0xffffffffffffffff,%rcx
400ab2:       48 89 85 20 ff ff ff mov     %rax,0xffffffffffffffff20(%rbp)
400ab9:       b8 00 00 00 00 mov     $0x0,%eax
400abe:       fc                       cld
400abf:       48 8b bd 20 ff ff ff mov     0xffffffffffffffff20(%rbp),%rdi
400ac6:       f2 ae                   repnz scas %es:(%rdi),%al
400ac8:       48 89 c8 mov     %rcx,%rax
400acb:       48 f7 d0 not     %rax
400ace:       48 8d 50 ff lea     0xffffffffffffffff(%rax),%rdx
400ad2:       48 8d b5 50 ff ff ff lea     0xffffffffffffffff50(%rbp),%rsi
400ad9:       48 8b 7d f0 mov     0xffffffffffffff0(%rbp),%rdi
400add:       e8 76 fb ff ff       callq 400658 <strncmp@plt>
Integer compare: 6 cycles

\[
equal = (\text{red_apple_six} == \text{inputNum});
\]

```
400d8b:  8b 45 b0                mov  0xffffffffffffffffb0(%rbp),%eax
400d8e:  83 f8 06                cmp   $0x6,%eax
400d91:  0f 94 c0                sete  %al
400d94:  0f b6 c0                movzbl  %al,%eax
400d97:  89 45 ac                mov   %eax,0xffffffffffffffac(%rbp)
```

Straight-line code, no function calls.

➤ For performance, prefer ints over strings if possible
One final example (Phew!): Limit testing
Limit testing for debugging

User wants to view ‘console’ of a VM

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: server problem
  - Server folks: client problem
  - Client folks: ‘vmrc’ problem (vmrc = VMware Remote Console)
  - VMRC folks: authentication? MKS tickets?
  - Me: Time to dig in...

- 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
  • Out-of-memory exception
  • Exception silently ignored (never returns to client)
No Console: Isolating the Problem

• Problem
  • VMRC requests 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

Important lessons learned
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
4. Be data-driven in your analysis 😊
Tying it all together...
Keep these things in mind as you design features
And figure out how to test them \textit{at scale}

Longevity: test the long pole

Tuning: optimize for the platform

Modeling: understand what performance \textit{should} look like

APIs: make them easy to use and hard to abuse

Back to Basics: reduce waste
An interesting link

Performance anti-patterns

http://queue.acm.org/detail.cfm?id=1117403

Some examples:

• Fixing Performance at the end of the project

• Algorithmic antipathy

• Focusing on what you can see rather than the problem

• Not optimizing for the common case
Examples for a future talk 😊

Python, cProfile, and recv()

VMotion tuning for high-latency networks

DB IO tuning

Incorrect error handling causing CPU to gradually increase

Racing app threads preventing DB from making progress