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 moves a Virtual Machine while the Virtual Machine is still running. This slide (the PPT, at least) contain a video of a VM being migrated while it is playing a video. The video continues to run even when the VM it is running on has been moved to a different host.

Sree is my colleague – he created the video.
Brief overview of Compute vMotion. The assumption is that memory doesn’t change too often, so you can ‘pre-copy’ memory while the VM is still running, rather than stopping the VM entirely in order to transfer all of the memory while the VM is paused.

1. Pre-copy memory over network (CPU & Network) while the VM is still running. Keep track of changes and when you are done with one iteration, go back and do it again on the changed pages. When the changed pages is below a certain threshold, go to step 2.
2. Stun VM (i.e., pause the VM).
3. Copy remaining memory + checkpointed state (CPU & Network)
4. Switch VM disk pointer (via shared storage).
5. Re-start the VM.
Storage vMotion: move a VM’s disk from one storage device to another. This is in contrast to compute vMotion (described on the previous slide), which moves a VM from one host to another, where the hosts are sharing storage. In Compute vMotion, you do NOT have the move the disk of a VM.
For Storage vMotion, you precopy the disk (in contrast to precopying memory in the Compute vMotion case).

Intuitively, this chart shows the number of precopy iterations on the X-Axis, and the Y-Axis shows the time it takes to complete an iteration.

You want the latency to go down as you do more precopy phases. Each line is a ratio of transfer time to dirtying time. For example, the line at 100 (“1”) is horizontal because you are dirtying memory at the same rate you are transferring it, so each iteration takes the same amount of time, and you’ll never converge.

You want to be at the lower left quadrant, else you might not converge.

Because you may not converge, you need to approach this problem differently, as discussed in the next slide.
Here is the way we actually solved the problem.

a) Hypervisor does a linear pass through the disk and copies data a block at a time from source to destination.

b) When a guest does an IO, if it is to a block that the hypervisor has already copied, then mirror the IO to the destination storage. If it is to a block that has not yet been copied, then you do not have to mirror the write: when the hypervisor copies the block later, the change will be picked up. If the write is to a block that is currently being copied from source to destination, then delay the guest write until the hypervisor finishes copying the block.
Stories from my job
On a previous call with this customer, I asked if I could install a tool that monitors database transactions. They encountered a crash 7 days later (due to a DB spike in traffic caused by a cleanup job that runs every 7 days). Our internal testing had a requirement that you need to be up for 7 days, but it had crashed at 6 days, 23 hours, and 56 minutes (due to something unrelated...maybe running out of disk space or something), so we never saw this database spike.

Lesson: do ‘long-pole’ testing (i.e., run for a long time to make sure your server software doesn’t break); also, make sure that if some cleanup happens at, say, 7 days, either run it for 7 days so you can see what happens, or modify the code to make it happen more frequently so you can test for less time and still see the impact.
The right tool for the job...
We have a test that registers hosts to a central management server ("add host"). The new build was slower than the old build. A colleague saw some new messages in the logs, which helped us know what was newly added and might have caused the issue.
Yes, we see new log messages due to new code, and we think that is the reason for the delay. We are doing extra work. We might consider looking at the CPU to see if it is saturated. If so, any extra work could cause our benchmark to be delayed. Otherwise, if there is extra headroom, then maybe this work can occur in a different thread on a different CPU and would not account for the delay.
I looked at the code pointed at by the log messages Erin provided. These pointed to Remediate...(). The developers had a good point that this new code is running in a different thread, so it _might_ not be the issue.
The CacheUpdate() occurs in both cases (it is a no-op in the good case, since the functionality was NOT enabled in the good case). In the bad case, the code is enabled and is scheduled to run immediately.
CPU is not saturated in either the good case or the bad case, so just adding more work (in the new case) may not explain the issue – the new work could have been done on a different CPU. That was the developers point.
Using the RotateRight Zoom profiler (which is now end of life), we can see that in the bad case (AFTER), we spend more time in locks than in the good (BEFORE) case. So maybe locking is the issue.
If we enable extra logging, we can see that some locks are more expensive in the bad case vs. the good case. So something is causing locks to take longer to be acquired.

Logs are great, but they can slow things down, and you have to recompile your application whenever you need to add a new one. That’s why using tools is typically a better approach. It’s like debugging using a debugger (e.g., GDB) vs. printf: using a debugger is typically a better idea because you get deeper visibility than with printf.
I highly recommend buying these books. They are great for explaining how to do performance analysis and engineering.

The books I referenced in the previous slide discuss a lot of great tools for performance analysis. One example is BCC tools, built upon BPF (BPF stands for Berkeley Packet Filter, because these scripts and this language used to be primarily employed for network observability).
Runqlat tells you how much time you spend waiting to run tasks. You can get it in a per process basis.
Runqlen tells you how busy a CPU is. It’s like CPU occupancy. Lower means the CPU isn’t saturated, but this is just a snapshot in time. You may wish to graph it over time to see how it behaves overall.
Here, I graph the runqlen over time. It looks like the bad build (afterNKP) has a longer runqlen (more tasks queued up), but as you see on the next slide, there is run-to-run variation.
You can see here that in this case, the good run (beforeNKP) actually had a longer runq. So this data doesn’t necessarily explain the issue.
Flamescope is another great tool. Refer to the links above to learn more. In the chart above, the x-axis is time in seconds. The y-axis shows utilization for every 100ms in that second. Each square is 100 ms. You can thus see how busy you are on a sub-second basis.
The flamescope plot in the ‘after’ case (the bad case) suggests that we are, indeed, doing more work (there is more red over time).
When comparing before and after, the ‘after’ plot looks busier, which may explain why it is slower. There is more to this story, but I’ll ignore it for the sake of this discussion.
The next few slides are snippets from a script I used to measure the time to add a host and the time spent in various locks. I create a histogram of both. For more info on the syntax, refer to the links in the books I mentioned.
BPF script for tracking locks and add host calls (2/3)

```c
uprobe:/usr/lib/vmware-vpx/vpxd:ClusterMvMAddHostInt
  /$1 == 0 || pid == $1/
  {
    @addhost_start[tid] = nsecs;
    @addhost_add[tid] = @addhost_add[tid] + 1;
    @addhost_started = 1;
  }

Do stuff on function exit
```
Bpf script for tracking locks and add host calls (3/3)

1. Compute latency
2. Store in histogram
3. Cleanup

```c
uretprobe:/usr/lib/vmware-vpx/vpxd:ClusterMcuAddHostInt
(/$1 == 0 || pid == $1) && (addhost_start[tid]){
    @addhost_latency_ms =
    hist(1000000 - @addhost_start[tid]) / 1000000;
    @addhostcount = @addhostcount + 1;

    delete[@addhost_start[tid]];
    delete[@addhost_addr[tid]];
    if (@addhostcount == 2) {
        exit();
    }
}

clear[@lock_start];
clear[@addhost_start];
clear[@addhost_addr];
```
The output shows a histogram on the left and the number of entries on the right. For example, you can see that 13 ‘addhost’ operations took between 4k and 8k ms (4-8s), while 51 of them took between 8s and 16s.
There were a bunch of locks that only showed up in the new (slower) build. In one case, we were grabbing 1266 such locks. Although most of them took very little time (they were in the bin for “0” ms), they still impact performance, as I explain in the next slide.
The application had some hierarchical locks that required grabbing every lock in a tree. The locks on the previous page were leaf-level locks, so even if they individually take very little time, if you are grabbing them very frequently, they can prevent the hierarchical (EXCLUSIVEALL or SHAREALL) locks from making forward progress. Ultimately, it was this sort of delay that caused our performance to regress. The new code grabbed these leaf-level locks and slowed down the main application, which needed the hierarchical locks.
I’d encourage to get used to using tools other than log statements, since you can get deeper visibility without recompilation, and you can also examine network, disk, etc. (though the previous examples focused on CPU).
What about Memory?
Chap is a tool written by my colleague. It analyzes core dumps.
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

Chap is a great complement to GDB.
This picture is taken from the documentation.
The command “count allocation Orange /extend ->” will find every memory object of type ‘Orange’ and extend the list with anything ultimately reachable by Orange. This list includes all of the numbered items in the picture.
If chap cannot figure out the data type, it calls the item "unsigned," and anything of the same size gets lumped together. If chap can figure out the data type, it prints it out.

"summarize used" will summarize all memory that was currently in use at the time the core dump was taken.

Chap is often useful as a comparison tool. Run it on something that behaves well, and then compare against a case where you have a memory issue.
Free memory includes memory that is not in use by the process, but has not been reclaimed by the OS. If you have a lot of free memory, it could be that you are seeing significant memory fragmentation (so the memory cannot be re-used). In cases like this, a good heap allocator is key.

You can also list every element of a certain size, so that you can examine those addresses in gdb to find out what they are pointing at.
Useful chap commands

chap> describe used - /size 48 /extend <==>StopHere /commentExtensions true
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.

You can also get more information about elements of a certain size using the 'describe' command.
In our case, we had an app talking to a database, and the app ran out of memory. Inspecting dhap, we realized that the DB could not keep up with our app, so the app kept accumulating data.

```
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

Lesson:
1. Understand the queues in your system!
2. Do Long-pole tests
```

© 2019 VMware, Inc.
I mentioned before that the Zoom Profiler is end-of-life (EOL). I’ll briefly discuss the Linux perf utility, which is also a sampling profiler. It is well-supported by the Linux community, and it is free, so I recommend learning how to use it.

I decided to use perf on the chap utility I just described.
I downloaded the chap utility, compiled it with debug symbols, and ran it on a core dump. It was kind of slow (in fairness, I was analyzing a 30GB core dump, which is quite large), so I wanted to know what was going on. First, I used ‘top’. You can see that only one CPU is occupied, and it is occupied by chap. So, I decided to use the Linux Perf utility.
Here are the commands to
a) record a profile for a process named ‘vpxd’ at a sampling frequency of 99 samples per second, with call graphs, running for 60s, and
b) creating a report that include call graphs (as seen on the next page).

```bash
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
```
This is the overview chart, showing CPU breakdown by module. You can dig in deeper by clicking on modules with “+” next to them.

% perf record -call-graph dwarf -F 99 -p `pidof my-chap` -- sleep 60
% perf report -i perf.data -g
You can see from this stack trace that _start calls _libc_start_main, which calls main. Then, main calls ELFCore64FileAnalyzerFactor::MakeFileAnalyzer, which calls ELFCoreFileAnalyzer. The ELFCoreFileAnalyzer calls LinuxProcessImage, where a lot of the real work happens when loading in a core dump.

You can dig deeper to see more of what is going on.

What I’d do is run perf on a ‘good’ run and then run it on a ‘bad’ run and compare to see how time is spent differently in each case.
Interesting bugs or observations
A virtual machine is basically a file stored on disk plus runtime state that executes on the CPU (plus memory).

When you clone a VM, you basically copy the disk file somewhere else and initialize a configuration file. When you start the VM, it has the same setup as the one you cloned.

As an example, maybe you've created a VM for your class, and it has a bunch of libraries and debugging packages. If you clone it, then each of your students can get a VM that already has all of that stuff pre-loaded.

In this talk, I will assume that the source VM is powered off when you clone it, so you don’t have to copy any runtime state like memory.

In this case, if a disk file is located on local storage, then cloning it from S1 to a shared SAN so that it can run on D1 should involve reading the data into the memory of S1 and writing it do the SAN.
I noticed an interesting asymmetry in performance. If the source VM had its disk on a local disk, and if I was cloning it to a shared SAN, this operation was slower than the reverse (namely, copying from SAN to local storage). Why?
It was a bug. I will explain in the next few slides.
The VM v1 is running on S1 and has its disk on the shared SAN (SAN = storage area network, basically a group of disks you can access over a dedicated network like Fibrechannel).

When cloning it to the local disk of D1, here is what happens: D1 reads the disk data into its memory and writes it into its local drive. This is pretty fast.

In essence, the destination (D1) is driving the clone, and it has access both to the source and the destination disk, so it reads data into memory and writes it to disk.
Now, consider the case in which the VM v1 is running on the local storage of host S1 and needs to be cloned to the shared SAN so it can run on D1.

You would expect that S1, which can access both the source (local storage) plus the SAN, would simply read v1 into its local memory and then write it to the SAN, and would be as fast as the reverse case (SAN to local).

However, there was a bug in the code. In the code, the destination host was always the driver of the clone. So D1 did the cloning. D1 does not have direct access to the local disk of S1, so D1 reads the data over the network and then writes it to the shared SAN. Reading over the network is slower than a read directly from disk (in this case).

The fix was to do a check at the start of the clone: if either host can access both the source device and the destination device (i.e., if either host is connected both to local storage where v1 is located and the SAN), then that host would do the clone. In this case, the test would determine that S1 can access both local storage as well as the SAN, so S1 would drive the clone, and we’d get equal performance as the SAN-to-local case.
Different platforms (OSes, Databases, etc.) have different performance characteristics, so when you make changes to a platform (like upgrading the OS), you may need to do profiling to make sure it performs as you expect.
In this case, the x-axis is time, and the y-axis is performance (higher is better).

We were using SLES (SuSE Linux Enterprise Server, similar to Red Hat or Ubuntu Linux), version 11, and we saw good performance.

We upgraded to SLES 12, and we saw a drop in performance.

We were also looking to move to our own distribution (Photon7), and this also had the same poor performance.

We made a bunch of changes to Photon (Photon 8), and we get back our performance. Let’s understand why.
When we first saw problems in SLES12 and Photon7, we asked some of our kernel folks to take a look. They made a bunch of changes put them in Photon8, and gave us the config file. We used it and got good performance. However, it is extremely important to understand what changes were actually necessary, since some of the new features (like stack protector) may have security implications if you disable them.

I went through each of the changes they made and ran a test with only those changes. I finally determined which changes was responsible for the performance boost in Photon 8.
Initially, SLES11 did well, SLES12 did poorly, and Photon8 did well. Recall, Photon8 had a bunch of kernel changes, and some of those kernel settings were also present in SLES12.
The changes in Photon8 involved disabling a bunch of new settings. When I successively re-enabled them, re-enabling JUST NUMA_BALANCING caused the performance drop.
Once I disabled NUMA_BALANCING, I regained the performance lost while moving to SLES12 and Photon7.

NUMA_BALANCING was newly introduced in the Linux kernel as of 3.8.0. SLES11 didn’t have it, because it was at 3.0. SLES12 and Photon both had it, because they were at 3.12 and 4.2.0.

This setting effects memory balancing. NUMA == Non-Uniform Memory Access, for example, if a CPU in socket 0 is accessing memory from socket 1, that will be slower than if the CPU in socket 0 were accessing memory in socket 0.

If you enable NUMA_BALANCING, then the guest OS (SLES12 or Photon) will try to move memory to be closer the the CPU. However, since we are running in a virtualized environment, the virtual CPU might be running on a different socket from the memory, so you may actually be making memory behavior worse by trying to balance within the guest. Instead, disable it and let the hypervisor manage memory.

It is important when switching platforms (or upgrading) to read the release notes. It sounds kind of mundane, but you will have a heads-up as to things that may alter performance.
It is good to make a performance model so you can understand how you _think_ something will behave. Then, if the behavior doesn’t match your intuition (your model), you need to investigate.
In this benchmark, we want to tell a central management server (vCenter) about newly-added VMs in the system. Basically, we have a script that tells vCenter that a new disk file is present on the host (ESX), and vCenter adds that disk as a VM to its inventory of VMs. This process is called VM registration.

We expect each VM registration to probably take the same time, so the end-to-end time _should_ be linear in the number of VMs. But it wasn’t...
The pink line shows the original performance. Instead of linear, it is quadratic. I'll explain why in the next slide.
The yellow line shows an update release where we fixed the issue and made it much closer to linear.
The blue line shows an additional fix.
The reason the `registerVM` call was quadratic is that each call was checking to see if the disk file handle was unique. This was done by linearly traversing the list of file handles. So the Nth VM registered would need to examine N-1 file handles, an N-squared operation.

This was a known bug, but before I generated data to show the performance impact, it had been deferred. They decided to fix it after seeing this data.
APIs

Toolkits and APIs: pitfalls and perils
PowerCLI allows you to interact with a virtual machine management server using shell-like commands. In this case, we want to get the number of virtual CPUs for every VM in the system. We look for each VM (Get-VM) on a given host (-Location $esx).

The trick here, though, is that we do client-side filtering: each call to Get-VM gets every VM in the system, and the client then filters them out based on which host they are running on. If the server were written differently, it might have enabled the query to do server-side filtering, and only return the VMs running on that host.
Pulling Get-VM and Get-Vmhost out of the loop dramatically improved performance (you get all VMs and hosts once, and you iterate over them).

There are many things to keep in mind when creating an API.
1. How will it be used?
2. Do you need a unique identifier? Can it change?
3. Can you query by name? What if the name changes? Do you have a notification API so users can monitor for changes?
4. Also, when you provide code samples, be aware that someone might use them in their production code, so make sure they scale, or give instructions about how to make them scale.
A customer had a management VM whose memory grew so much that the VM crashed. I had seen this many times before, and I based on experience, I suspected a misuse of a monitoring API.
The code above shows how to get every host and VM in the system and print out their names. If you call the code above repeatedly and do NOT do DestroyView(), then you will keep memory around in the system. The rule is that when you are done getting the information you need, if you don’t reuse the ContainerView, then destroy it.
I checked some instrumentation we have in our product, and sure enough, this was what the customer’s code was doing. We also had a similar problem with some products that talked to our product, so we had to talk to those partners about fixing the code.

The bottom line is to educate customers about your APIs, provide solid code samples, and also consider how to make your platform resilient (for example, maybe you detect multiple creates without a destroy, or maybe you detect that something hasn’t been used for a while).
Learn to use monitoring tools so you can easily view time series data. My previous charts were using Excel, which is less feature rich than other options like Grafana, Wavefront, etc.
Back to Basics
Java and C++
Most of you know how Java memory management works. To start with, there is reachable memory, and there is unreachable memory.
Memory is actually code cache + heap + permgen + thread stacks. A garbage collector finds unreachable memory, deallocates it, and compacts the existing memory.
If an object is newly created, it goes into the Eden region. If it survives a GC (Garbage Collection) cycle, it goes to Survivor. If it survives repeated GCs of Survivor, it is considered long lived, and goes to OldGen. OldGen GCs occur less frequently and take more time than other GCs.
Learn about GC settings for the JVM you are using. You may need to tweak them for your use case.

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.
There is typically a tunable that says how much of the heap should be kept free. If you exceed this, you do GC more aggressively.

In our case, we had seen OldGen grow and grow and grow, until it took around 20GB. This was causing contention for memory for other processes in the system. We had set min heap to max heap, which means that the JVM lets the heap grow until it reaches the max. If we set min lower, there is more incentive to GC and reduce memory usage, but the GCs might be costly.
We profiled our app more and realized we didn’t need as much heap. By shrinking the heap, we got better GC performance and consumed much less memory.
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
We sometimes use this routine to measure the latency of a procedure.

```c
RDTSC: read timestamp counter
Let's you see the number of cycles for a section of code

#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
```
A string comparison is much slower than an integer comparison. If you have a choice, use integer comparisons instead of string comparisons for better performance.

```c
Rdtsc string vs. integer compare

```t1 = rdtsc();
for (i = 0; i < num_iters; i++) {
    equal = strncmp(s1,s2,strlen(s1));
}
t2 = rdtsc();
```

String comparison: 81 cycles per loop

```c
for (i = 0; i < num_iters; i++) {
    equal = (num1 == num2);
}
t2 = rdtsc();
```

Integer comparison: 6 cycles per loop
Strncmp: 81 cycles

% objdump -S -C test

```c
equal = strncmp(s1, s2, strlen(s1));
```
Integer compare: 6 cycles

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

```
400d8b: 8b 45 b0       mov  0xffffffffffffffb0(\%rbp),\%eax
400d8e: 83 f8 06       cmp  $0x6,\%eax
400d91: 0f 94 c0       sete \%al
400d94: 0f b6 c0       movzb \%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
When a customer wants to view the console of a VM (in the physical world, this is like going up to a terminal that is connected to a physical machine and logging in), they connect to the management server, which finds the VM and then establishes a session between the user and the VM.
My colleague noticed that with a sufficiently large number of hosts running VMs, he wasn’t able to view the remote console: it was blank.

I did some incremental testing and found that the problem happened with 1001 hosts. The problem wasn’t that someone used < instead of <= or something like that…it was a bit more subtle.
To debug, I observed the console and the resource usage on the management server at the same time. In the normal case (when things work), the management server sees a spike in CPU in memory right before the console appears.

In the case that didn’t work, the management server hit an exception, and the remote console never showed up.

When I attached a debugger, I saw that the exception was an out of memory (OOM) exception, and it was silently ignored, so the client never showed up.
Why was there a spike in memory before the console appears? And why did we hit an OOM, thus preventing the console from coming up?

The management server gets info about a VM before showing the console (e.g., is there a CD-ROM attached?). However, the developer get info for EVERY VM in the system, not just the VM that the user wanted to view. With a large enough environment, this request overflowed a buffer and caused an OOM. The OOM wasn’t handled properly, so we never got a console.

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 at scale

Longevity: test the long pole

Tuning: optimize for the platform

Modeling: understand what performance _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