PROFESSOR: And so today we have Ravi , who is I guess a performance engineer at VMware, where he basically looks at how to get really good performance in many, many different things in VMware. So Ravi is an MIT undergrad. I guess a long time ago.

RAVI SOUNDARARAJAN: Yeah, '92.

PROFESSOR: '92. There you go. And then he got his PhD from Stanford and end up in VMware looking at a lot of interesting performance stuff. So today, he's going to give us a kind of a bird's eye view of how these kind of technologies, what you guys are learning, is useful and how it's used in industry. OK, Ravi. Take over.

RAVI SOUNDARARAJAN: First question, can everybody hear me? And is the mic on? OK. Good. I have a bit of a sore throat. So I'm sorry if you can't hear me.

The other thing I want to say is I'm actually really glad you're here. Because I remember very distinctly in 1990 when I was an undergrad, I was taking 6.111. And I had a bug in my final project. It just wasn't working. And I was going to skip the lecture that day. But I was like, whatever, I'll go.

So went to the lecture. Turns out Don Troxell pointed out something in that lecture. I was like, that's my problem. And I went back, fixed it. Everything worked. And I was very happy. I don't think that's going to happen here. But it would be nice if it did. So that's cool.

So instead of just talking about what a performance engineer does, I would-- I'll just-- Power is coming in. But-- Oh, there it is. Perfect. So instead of just talking about what performance engineering is about. I just wanted to go through 10 bugs

that I've actually been involved in. And hopefully, they'll be interesting to you. Because each one of them was interesting to me.

So let's start out with-- I just wanted to start this out with a bang. So for dramatic effect everybody read through this slide. It's basically an email thread from one of my colleagues. And raise your hand when you're done. So I'll know when you're done.

OK. Anyway, so the point of this is that his question ended up being, why is the percent lock time in the database increasing despite a lighter load for the thing that uses that database? Now, if you're like me, you read this email, and your first question is what the hell is he talking about? So that's why titled this slide like that.

So let's actually deconstruct what, a, that email meant. And, b, what it meant to me. So in order to do that, I want to first talk about the system that he was talking about and help describe what the actual problem he was seeing was. So here we have a standard multi-tier set up. What we've got is that we've got a server, which is our middleware server. And it talks to a database. And then we have a bunch of hosts on the other end. Those are labeled by agent, HAL, whatever. Anyway, those are hosts running virtual machines.

And here's the setup. You've got a client. And that client issues a command to this middleware server. And that middleware server tries to figure out what host or what virtual machine needs to be contacted. So the middleware goes ahead and does that. And it performs an operation on that host. And after performing the operation, it then archives some of the information that it gained to the database. And once it archived the information to the database, it went ahead and notified the client that the operation itself was done.

Fairly simple. Send a request to a server. Server contacts some other host. Host comes back to the server and says it's done. Server persists information to the database and then notifies the client everything is done.

So the actual problem that my colleague was seeing was, he was sending less and

less load to this middleware server, which therefore means that less and less goes to the host and less and less goes to the database. However, the percent of time spent in the database was higher. So despite a lighter load and less stuff going to the database, the percent of time spent in the database was higher. Probably, all of you are smart enough to know what the actual problem is. But we'll go through it step by step.

So the deal is this. The first thing I did is I examined the amount of time that each lock was held for each of these different operations. And what I did was I varied the load just as my colleague did. And what you see are six different lines. On the x-axis is each of the different locks that we acquire in our system. And on the y-axis is the latency of those locks in milliseconds. And basically, the axis for this-- the legend for this chart says Locked-4, which means a certain type of load, all the way down to Locked-128.

OK. And the deal is that when you have lock-128, what that indicates is that there's 128 hosts per thread. Which, take my word for it, that means that you've got a lighter load. So basically, the lowest line in the legend is the lightest load. And the highest line in the legend is the highest load. And as you can see, although it's a little bit obscured in this chart, the bottom line is that this is about time you're spent holding a lock.

In this chart, lock-4 indicates a heavy load. Lock-128 indicates a lighter load. And what this chart indicates is that, in fact, you are spending more time in the locks with a heavier load, lock-4, than you are with a lighter load, lock-128. So the latency prologue is higher with a higher load, which is totally reasonable. And it turns out that if we translate my colleague's question into reality, what he's asking is why is the percent time spent in the database higher, even though you have a lighter load?

Well, the answer ends up being that when you have a lighter load, overall, you're spending less time in locks, which means that any time you spend in the database, if that's a fixed cost, is going to be a higher percent of the overall time. So it turns

out, there really wasn't an issue. There was nothing odd here. It's exactly what you'd expect, because the database imposes essentially a fixed amount of latency per operation.

And in order to confirm that, I also looked at the time spent contending for locks. So you really want a lock, but someone else gets it instead. And you have to wait. And the trend is exactly the same. When you have a lighter load, you end up having a much lower overall latency. But that just means that any latency you spend in this database, its percent impact on the entire operation is quite a bit higher.

So to take a look at this and to do a postmortem on this case study, his first thing was some random email, which kind of was nonsense. But it was less nonsensical once you actually tried to understand what this setup was. So step one is figure out what is the experimental setup and what are the parameters that are being studied. The next thing to understand is what's being measured? I remember what my colleague said. He said the percent of time in the database was pretty high. Why is that? Well, OK. So we have to understand what this multi-tier setup is, why you would be going to the database, and what it means if the time spent in that database is high.

And I guess the next thing you want to do is you want to say, well, OK, he's complaining about lock latency. So I must have some sort of visibility there somewhere. Let me go and look at the appropriate metric to figure out what's going on and why time is being spent there.

And finally, it's appropriate to-- once you've got all this data, don't make any assumptions-- just draw the appropriate conclusion based on the data. The problem is basically that with a lighter load the impact of that database becomes larger, even if it were the same under a heavier load. Because under a heavier load other aspects of the system also get stressed that don't get stressed in a lighter load.

So, and this is basically what I'm saying here, yeah, the percent lock time in the

database overall is higher. But the overall lock time is smaller. So everything is behaving exactly as you would expect.

So this was one simple study and hopefully it didn't require any virtualization specific knowledge. What I'm going to do now is go over a bunch of other different case studies that I've looked at and crystallize what lessons I learned from doing those studies. So I guess the next case study I want to talk about is something that I call garbage in, garbage out. I think that's something I learned on the first day of 001.

So here's the deal. You've got a customer. And he wants to draw this chart. And this chart is basically, on the x-axis is time and on the y-axis is some metric. It happens to be CPU usage megahertz. But it doesn't really matter.

OK. So he wants to get this chart. And it turns out-- if you remember the diagram I showed before, we have a server and a bunch of hosts at the end. So the client simply has to send a request to that server. And the server will generate the data, send back to the client. And it turns out that if any of-- all of you, I'm sure, are familiar with Pearl, Python, and all these different scripted languages. There's another one that's very popular in the Windows community called PowerShell.

And so it turns out that our server has a PowerShell interface. You can write a PowerShell script to get any statistic that you want. And then you can write that to some database or some spreadsheet and get this charge.

So here's the deal. We have a user. And what he does, as I've indicated at the bottom, is that he wants to get the CPU usage for the last hour. So he just writes a simple script, using PowerShell. It doesn't really matter what the syntax is. But the point is, he does what's called a get-stats call. And he indicates, OK, I want this virtual machine. I want this stat, whatever. And then he gets his information and writes it to a file.

No big deal. It's just a simple script to get some statistics. Now, here's what is actually interesting. I wrote that script using PowerShell. Now, let's compare a

PowerShell implementation to a highly tuned Java implementation of the very same script.

Now, what I've done here is I've shown the performance of that very same script when getting data for one virtual machine, which is the first row, six, about 40. And then up to a little over 300 virtual machines. Look at the scaling of this script. In the middle column, I have the performance for this PowerCLI script that I just showed you on the previous slide. As you can see, it goes from a healthy 9.2 seconds to over 43 minutes if you increase the size of your inventory. In contrast, if you use a Java-based implementation, it starts a little slower than PowerCLI at 14 seconds. But the scaling is far superior. It ends up at 50 seconds.

So the question is why? What's different about these scripts at scale versus when I wrote the little toy script? Now, to give you some color, why this matters to someone like me is that every once in a while, I'll get a call from a system admin, who wants a very simple script that does blah. And they say, I wrote this simple script in my development environment, and it worked fine. But then I run it in my production environment, and it completely breaks. What's going on?

So often, problems like this actually occur. Now I'm calling this garbage in, garbage out, because you can see that with PowerCLI and Java, we're getting very different results. The deal is that in PowerCLI, I wrote a very naive script. And I'll explain how in a moment. And that very naive script has not performed at all. Whereas in Java, I wrote a very highly tuned script. And it ended up taking about an order of magnitude less time.

So garbage in, garbage out. Good stuff in, good stuff out. Let's try to understand why. OK. Here's the deal. All of you are probably familiar with the difference between scripting and high level languages, or low level languages or whatever you want to call them.

So the idea is that if you're running something using a shell script or running Pearl, or whatever. These things are usually meant for ease of use. But they're not necessarily meant for performance reasons. And so they hide a lot of details from

the users so that it's easier to come to a quick prototype. You don't have to declare variables. Do all this kind of junk, right, if you're using scripting language.

In contrast, if you're using, Java, C++, whatever, you have Typesafe and all this nonsense. You have to worry about what the declarations of your variables are. Make sure they're correct types, et cetera. So they tend to be more difficult to get running. But once you get them running, you can use a lot of advanced tools that just aren't available necessarily in toolkits.

And we're going to talk about how this difference manifested itself in this particular problem. I said that when I told the customer to write a simple PowerCLI script, they just have to do one call. Get-stat. And here's a bunch of arguments to tell it what statistic it wants to grab. It's a toolkit. It's doing a lot of work for you.

What's it doing for you? Well, it turns out it's calling our internal APIs. Our internal APIs have to work for a variety of different platforms. So they have to be incredibly general. Under the covers of that single get-stat call, we have about seven different steps that are going on, ranging from asking, OK, I want a stat for a virtual machine, what kinds of stats does that virtual machine export? How frequently does it update? All kinds of information that you probably don't really care about if all you want is a single stat. But you might care about if you're writing a production quality script.

So the point is get-stat does these seven calls. And because you're using a toolkit, it hides them all from you. It spares it. It's very easy to use. But it turns out, this is obviously not the most optimal way to do this. And I illustrate that by showing you conceptually what's actually going on.

On the right, I guess on your left, I've got what the PowerCLI script was doing. You call this get-stat call, or whatever, and what it does that for every virtual machine that you want this statistic for, it goes through and does all these random steps that were done that I described in the previous slide. Figuring out what stats are supported, how frequently they updated, and then actually going and getting the stat.

It turns out, as I show on the right, when I do this in Java, I can take advantage of the fact that a lot of those things that I'm asking about in the toolkit, they never change. You've got a good virtual machine. It supports things to be refreshed every 20 seconds or whatever. That never changes. You want to find out what statistics are supported. Well, those never change once you've configured it and you don't change anything. So why do that every single time?

So as we show in Java, you can pull a lot of that stuff out of the main body of the loop. And if you wanted information for every VM in the system, you only have to make one call, as I show at the very bottom, for every VM. You don't have to make seven different RPC calls. In fact, you could further optimize, if you really understand the API, and collapse a few more calls in Java that you can't do in the toolkit. Because again, the toolkit's trying to talk at a high level, so any random dude can actually write a script.

OK. So the point is in this PowerCLI world, a very simple toolkit world, we were making five RPC calls for every VM that we wanted stats for. Whereas in Java, we made one call. And we can even optimize that further. And I don't show that here. But anyway, the bottom line is that's why the performance of the Java script was so much faster.

So the idea is that with PowerCLI what I do is I wrote very simple, a script that anybody could understand and use, but it's not optimized. I didn't utilize multi-threading. I didn't realize that the output format of the data was really, really verbose. Again, that's the kind of thing that if you really understand the APIs you can take advantage of.

I also didn't realize-- as a customer, I wouldn't have realized that I make this one call but it expands it to a bunch of different network requests that I didn't know about. In contrast, with Java, I was able to take advantage of the power of a language like Java, where I could thread pools. I could optimize the return format of the data. And I could reduce the number of RPC calls.

I guess the analogy I would give is if you think about assembly code versus compiler-generated code, nobody here wants to sit down and write x86 Assembly. So, of course, you write C and you use GCC and whatever. But maybe there are some things that would actually be faster if you wrote hand assembly.

OK. So, first example we gave was just understanding what on earth the problem was. The second example was if I write bad code, I'm going to usually get bad output. In the third case, I'm going to talk a little bit about the design of an API. And I'm going to motivate that by an example.

We have an administrator on the right hand side. And he's talking to the management server. This management server is responsible for telling that administrator what's going in every host in the system, every virtual machine, whatever. So let's pretend that the user wants-- so this is a virtualized environment-- so the idea is that the user wants to actually access his or her virtual machine.

They're running a Linux virtual machine. They want to see its console. So they can type at the console. Here's how that roughly works. They want to see the console of the VM that I've circled. So the way it works is that this user talks to the management server. The management server locates that virtual machine, establishes a handshaking connection. And that connection is directly given back to that user.

So now the user can interact directly with that virtual machine and send commands and do whatever. So we've all got the setup. Very simple. Now here's the deal. The problem was that our poor little user, poor big user, whatever, they could not start this console when they were managing a large number of hosts.

To take you back to this diagram for just a moment, on the right hand side of the screen, I show a number of hosts. And I show a number of virtual machines. So what I'm basically saying is that this administrator could not access the console of his virtual machine as the number of hosts increased. There should be no link between these two. But there was.

So I got involved in this problem. And here I went to the client folks. And I explained the problem. And their response was that's obviously a server problem. Next step, I go to the server folks. And they said, yeah, it's obviously a client problem. So then I went back and thought a little bit more about the architecture. And what I knew is that whenever the client needs to initiate this handshake connection, it has to spawn a certain process.

So that process is called the VMRC, VMware remote console. So I talked to the client folks again. And said, well, I think this is happening. Oh, yeah, it's a VMRC problem. OK. So that just passes the buck another way.

So then I talk to the VMRC folks. And I said, OK, I think I have this problem with starting up. And they said, oh, it's probably authentication. It's probably this. You have to enable verbose log in on your end host, blah, blah, blah.

And at some point, I got so sick of this. I thought, well, this is complete garbage. I'm actually going to take a look at this a little more carefully. And so, once I got involved, I got a little bit more information. And the information was actually pretty useful.

Here's the deal. When you're starting the remote console on one of these virtual machines and you want to connect, it turns out the person that was having this problem-- when you ask a few more questions, you get a few more answers. Some of which are actually useful.

So I said, well, if you do this with 50 hosts, do you see a problem? So 50 manage hosts, he only wants to view one VM's console. No problem. Then I asked him-- and then I actually sat at my desk, and we reproduced this. So then I said, let's try it with 500 hosts, no problem. Just a little bit slower, but no problem. So again, the set up is 500 different hosts, he just wants to look at one VM. And it's taking longer and longer to view that one VM.

Finally, he gets to 1,001 hosts. And that's when the problem occurs. That' when he cannot see this console. So obviously then, you go back and you think, well, gee,

is there some magic number of 1,000 hard coded anywhere, such that we exceeded it, and that was the problem. And frankly, the answer is no.

So let's figure out actually what did happen. So I made a bunch of different observations while I was debugging this. When you have fewer than that magic number of 1,001 hosts, here's what was going on.

Remember the sequence of events. I'm a client. I talked to a server. Server talks to the virtual machine. And then that virtual machine and I are directly connected.

What I was noticing was that when there were less than 1,000 overall hosts in this setup, when I would initiate the command for this console. The CPU usage and memory usage of this middle management server would gradually increase, and increase, and increase until the console would actually show up.

So you right click. You want to pull up a console. A little window comes up. But it's totally blank. And then eventually, after the management server consumes a lot of CPU and memory, eventually that console comes up. So now your question is what's going on that's causing the CPU usage of this management server. And why is that scaling with the number of your virtual machines.

So the first thing I did-- so one of the themes of this talk, I suppose, is that there's a lot of great tools for performance debugging. But sometimes there's no real substitute for common sense and for the things that are easiest to do. So the first thing I did is I looked at the log file of this server. Here's the rationale.

If every time I invoke this remote console, I'm doing more work on the server. First, let me just see what that server is doing. Now, it turns out that what I'd noticed about the server was that whenever the client would contact the server and say I want the console of this VM, you would see a data retrieval call from the client to the server. And that data retrieval call got more and more expensive with more hosts and more virtual machines in the system. So the question that I asked was, why on earth are we doing some stupid data retrieval call if all I wanted is to say, here's a VM. Tell me where it's located so I can talk to it.

So it turns out-- that was problem number one. Problem number two, that's what happened when you had 50 hosts, 500, whatever, that you basically spend more and more time in this data retrieval routine on the server. However, once you hit that magic number of 1,000, you would see this call start. But you would never see it end. And in fact, you wouldn't see it end. You'd see some exceptions somewhere. So that to me was a clue that I better take a look and see what's going on at that exception.

So here's what I did. Unfortunately, the exception gave me no information whatsoever. So I went ahead and I attached a debugger to the server. And when I did, I found out that it was an out of memory exception. So here's the deal. I'm a poor admin. I send a request to look at a console. For some reason, some data handshaking code takes place between the server and the client. It takes longer and longer with more hosts. Eventually, when I have too many hosts, that thing ran out of memory and silently failed.

And so my client's sitting around looking dumb with a totally blank screen, because it has no idea that its request basically got denied. So the question is why is that happening? And aha, this is where the aha moment occurred. I happen to know that one of the reasons in our setup, that one of the causes for this data retrieval, is that as a client, I want to know certain attributes of the virtual machine that I wanted to view.

For example, if I'm looking at the console for a virtual machine, I might care that that virtual machine has a CD-ROM connected. And I might care if that CD-ROM got disconnected or whatever. That's not something you'd necessarily associate with a virtual machine. But when I go to the server to connect to it, the server wants to give me that information so that the shell, which is showing me this console can appropriately update if the virtual machine gets disconnected from its CD-ROM, whatever.

Anyway, the point is I looked at the code that this magical VMRC routine was doing. And here's what it was doing. It was saying, look, I want to start a remote

console session. I need certain information like the CD-ROM connectivity. I'm going to go ahead and get that information for every host and every virtual machine in the system. And then once that's done, I'll go ahead and display the console for this one virtual machine that I care about.

Well, this is clearly just a stupid idea. So I called the guy up and I said, you realize this is stupid idea. And he said, well, why? It's not that expensive. Is it? And I was like, OK, we need to have a long discussion. And bring a priest in here. But that's a terrible idea.

So what we ended up doing is-- And so now-- OK, so that's the problem. The problem is we drew a huge data retrieval. None of which we need. Now why this 1,001 host thing?

Here's the deal. It turns out that in our code, we had-- for old style clients, of which this was an old style client-- we had a restriction. We said if you need to serialize more than 200 megabytes worth of data, we're not going to handle that. We're just going to fail. It turns out once we got 1,001 hosts, we slightly exceeded that 200 megabyte buffer size, we silently erred with an out of memory exception. And we never returned to the client. And everybody was unhappy.

So it had nothing to do with 1,000 hosts. It has to do with the amount of data you're actually sending. So luckily, there were several different fixes. So I told the VMRC folks, the guy that wrote this thing, I said, please, under no circumstances, should you get information for every single host if you only want it for this one guy. That's a terrible idea.

And then I went back to the server folks, and I said, you have to admit-- first of all, you have to fail correctly. And second, you have to admit sensible error messages, because this should not take the kind of expertise that it took to figure out. And thankfully, this made actually a pretty huge difference. So that was kind of cool.

So the lessons that I took away from this is if you're going to create an API, create an API that's very difficult to abuse and very easy to use. Not the opposite. These

poor VMRC folks, it's like a three line piece of code to get information for every VM. It's like eight or nine lines to get information for just that VM. But those five lines make a boatload of difference.

The other thing is that this was an entirely internal customer. In other words, it's just another group within VMware. Imagine you're some external guy that's using the same API. You're screwed. So we have to be better at educating the people that are using our API. And don't just throw it over the fence and say here's an API. Deal with it. There needs to be some interaction between them to make sure that this is the right API for its job.

This is the next case study I want to talk about. This one actually was found by a colleague of mine, who I'm going to abbreviate as RM. Very interesting example. For those of you that took the class last year, I'm recycling it. But it's a good example, so no worries.

So here's the deal. You've got a benchmark. And I've got two builds of a piece of software. When I run the benchmark against that piece of software, in case a, I was getting a throughput of 100 operations per minute. In case b, I was getting half the throughput. And what was the difference between those builds?

Well, the first build, the faster build, was a 32-bit executable running on 64-bit hardware. While in the second case, it was a 64-bit executable running on 64-bit hardware. Now if you're like me-- which hopefully you're not-- but if you're like me, you think yourself, well, there shouldn't be much difference between 32-bit and 64-bit. In fact, 64-bit should be a lot faster for a lot of reasons you're going to learn about in 004.

So the question is what's going on? I can distinctly remember when this check-in was done, somebody said, oh, we're making the switch over, but it shouldn't be a big deal. And foolishly, like a moron, I actually believed that person. Very bad idea. So this came back to haunt us for three months.

So what's the deal? Remember, a 32-bit executable running on 64-bit code was

faster than a 64-bit executable running on a 64-bit application. So the first thing we do-- first of all, you've got to find the right tool for the right job. In this case, we use a profiling tool called Xperf.

Xperf is a very powerful profiler, kind of like Vtune, or Valgrind, or Quantifier, or a lot of the other tools you've used. The nice thing is it's built into the OS. It runs on Windows 2008. And it's a sampling based profiler that has a lot of pretty cool attributes. If you have Windows 2008, I think you ought to just go and download it. It's free. And play with it.

This gives a lot of information like stack traces, caller/callee information, et cetera. It's potentially the perfect tool for this kind of job. What kind of output does it show?

In this case, remember the setup. The setup is I run build A. And I'm getting twice as much throughput as running build B. So let's take a look at the CPU usage, just as a first cut to see what's going on.

In the 64-bit case, which is what I showed here. I showed the output of this Xperf for 64-bit. What you can see, x-axis is just time. And y-axis is CPU usage. It's totally saturated. In the 64-bit case, we're completely saturated. Take my word for it, in the 32-bit case, you're not. And this accounts for the difference. So now let's dig deeper and find out why we're seeing this difference.

So first, we look at the sampling profiler. Now there's a lot of perils of using sampling profilers.

In this particular case, here's the deal. I've got a process. And that process spawns a bunch of threads. In this particular view, what you're seeing is where is the time being spent in this process. And as you can see, most of the time is spent at the root, which kind of makes sense. Because everything in some sense is a child of the root.

But the first thing our poor process does, it spawns a bunch of child threads. So clearly, all of the time is essentially going to be spent in the routine that calls all of these threads, because that's the first entry point. But unfortunately, that's

remarkably unhelpful. So the deal is we have to dig deeper and find out, where is this time actually being spent.

So like I said, just to kind of reiterate the point that I made on the previous slide. The point is that your entry point is spawning all the threads. And if that's kind of the starting point for accounting, you're not going to get any help if you're just looking at that starting point. You have to dig a little bit deeper. And so what I'm basically trying to say, is that even though this thing records stack traces, this stack happens to be the most popular stack. But it's not actually the problem. Because it's the most popular, because that's where everybody starts. So all of the time that is charged gets charged to that stack.

We have to look a little bit deeper. So in order to look a little bit deeper, let's think about the problem in a slightly different way. Suppose I've got my root. That's where everything starts. That's where all the threads are spawned, whatever. And suppose I have three paths to a tiny little function at the very end. Maybe I'm spending all of my time in the path that I've indicated.

But the problem is that-- suppose you're spending all of your time in that tiny little function, and suppose it's equally spent among these other three paths. Well, it's not any path that's kind of screwed up. It's that that tiny function is kind of messed up.

So you got to figure out why are you spending time in that tiny function? And is there something that you can do to fix that? So let's look from the opposite perspective. Let's look at the caller view. In this case, what I did was I looked at a bunch of different routines, a bunch of these different so-called tiny functions. And I looked at what was calling those tiny functions. It turns out that here's a call, ZwQueryVirtualMemory some tiny function. It happens to be-- it turns out that we're spending, as it indicates, 77% of our time in this tiny function.

And just to emphasize that this is not being called from the root. So you don't see it from your first glance. You can see that the number of times this tiny function was called from the root is 0.34%. So if you were just looking from the root and trying to

figure out what's being called from the root. It would be totally useless. Instead, you have to dig deeper. And see all the different functions that are being called. And find out where they're being called from.

So we know that this function, ZwQueryVirtualMemory, is taking like 70% of the CPU, which is an enormously large amount of CPU. We also know what's calling it are these two routines right here. This thing, this magical thing called RTDynamicCast and RTtypeid. So it turns out we have two paths that are going to this tiny function. And those two paths comprise most of the calls to this tiny function. And it turns out we're spending a boatload of time in this tiny function.

So let's dig deeper and find out what on earth these two routines are. What is RTtypeid? What is RTDynamicCast, whatever? So let's first look at RTDynamicCast. What I show here-- in contrast to the previous view. The previous view said, I have a tiny function. What are the people that call me? Let's look at the reverse. Let's go and start at one of these two routines, RTtypeid. And figure out what it's calling that is spending so much time, and ultimately leads to us calling that ZwQueryVirtualMemory or whatever.

And as you can see, this RTtypeid, it calls two functions primarily. One is incrementing a reference counter. And the other is decrementing a reference counter. Now you're thinking to yourself, incrementing a reference counter, decrementing a reference counter should be very simple, very low CPU, and basically should cost you nothing. And all of those three assumptions are dead wrong, unfortunately.

So let's try to understand a little bit as to why that's happening. Turns out RTtypeid is used in order to figure out the C++ type of an object. So we're trying to figure this out on the fly. And as I said, it's kind of mystifying that this routine, which calls our tiny function, that you're spending 39% overall in this routine in that tiny function. So let's figure out what's going on by looking at a simple example, which would be IncRef.

So it turns out if you're looking at this code, the dreadful line is the fact that we're

doing const type info reference tinfo equals typeid of *this. This is basically, you have to call RTtypeid to get this kind of information. And in order to do that, you need some runtime type information. This runtime type information has some pointers associated with it.

So here's the deal. Whether I'm running a 32-bit-- Remember, we even forgot the original problem, which is, gee, I'm 32-bit, 64-bit executable-- sorry 32-bit executable on 64-bit hardware. We were slower than a 64-bit-- we were faster than a 64-bit executable on 64-bit hardware. Let's try to figure out why.

Well, remember, every time I increment or decrement a reference counter, I get some type information by consulting this runtime type information. It's got a bunch of pointers. When I run a 32-bit executable on 64-bit hardware, those pointers are just raw 32-bit pointers. You just look them up, and you're done.

In 64-bit, pretty much the only difference is that instead of those pointers being actual pointers, they're 32-bit offsets. So you take the offset, you add it to the base address of the DLL of the executable or whatever, where that runtime typeid call is being made. And then you're done. Once you've added that offset, you get a 64-bit pointer. You look up that 64-bit pointer, everybody's happy.

So here's the deal. Remember, we said 32-bit's faster. 32-bit is just a pointer look up. And then you're done. 64-bit's slower. It's a pointer look up-- and it's an addition plus a pointer look up.

So I pause, because you're thinking to yourself, an addition plus a pointer look up versus a pointer look up. Does that mean that addition is slow? That sounds really stupid. And by the way, it is really stupid. It's not true.

So the deal is this. It's not that addition to create a 64-bit pointer is slow. The deal is that-- remember this 2-step process. You find out the base address of a module. And you add this offset to that base address.

Determining that base address is really, really slow. It turns out you have to call a bunch of different random routines that have to walk the list of loaded modules.

And you're doing this every single time you call this routine. And if you increment a reference whenever you do a memory allocation, you can imagine that this is going to consume a lot of CPU.

So it turns out, this is what's actually slow. And remember, that we started this off a little by saying, oh, we've got this weird function, ZwQueryVirtualMemory, which seems to be being called a lot from a bunch of different places. The reason it's called a lot is because all of these routines to get typeid information are calling this routine. So there's actually two solutions. And you can look on this blog to actually figure out that this is a known problem that Microsoft eventually fixed.

And it turns out, that the two solutions end up being that you can either statically compute that base address so you're not constantly relooking it up every single time. But there's actually a much simpler and stupider solution, which is, use the latest runtime library where they have actually fixed this problem.

So just to summarize what was going on. We were CPU saturated. We looked from the top down. We didn't notice anything. We looked from the bottom up. And we noticed that some bottom was being called a lot. We figured out where that bottom was being called by looking at caller/callee views.

Once we figured out where it was being called, we then tried to figure out why that thing was being called. And then we fixed the problem. So to me, I guess the take-home lesson from this is that when that developer said to me, three months before this whole debacle started, he said, oh, yeah, we're switching from say 32 to 64-bit. That's not a big deal. You can't really take that stuff at face value, because a lot of little things can really add up. And we could never have really guessed that this would happen beforehand.

So that's four. Let me go over another one. Memory usage. So excessive-- I'm sure actually all of you know that memory usage is a big problem. Your application is going to slow down if you're inducing a lot of paging. And at least in 32-bit Windows, if you exceed two gigabytes for the process that you're running, the

process is just going to crash. So you clearly do care about memory usage.

And I want to differentiate between memory leaks and memory accumulation. You're wondering why would a process take up two gigabytes of memory. Well, one is that maybe you have a leak. You allocate something. You don't remove it.

That might just be an accumulation. You allocate something somewhere in your code, you actually are freeing it. But the trouble is that routine never gets called, because in your code logic you never take the time to think and say, OK, can I actually get rid of this at this point. And that's what I'm going to call a memory accumulation.

There's a lot of different tools out there to analyze memory usage. And the sad part about this is that almost none of them work for the particular example that we have, because they just didn't scale very well. It's fine if you've got your toy, stupid little application that does addition in a loop. Not so fine if you've got an enterprising middleware application.

So we ended up, despite all the tools that we have available, we ended up having to write our own. I didn't do this, a colleague did. But anyway, I guess that distinction doesn't really matter. But anyway, the point is that you do a lot of what these other tools do. But you do it special to your environment.

In this case, we just hooked a lot of the calls to malloc. And we figure out what pointers are live. And then we do stuff from there. This is how normal tools work. And of course, this can be unusably slow if you do a ton of allocations.

For example, if you're doing millions of allocations per second, you can imagine you're calling this kind of routine, which itself-- if it only has a 10x overhead over a call without that, you're slowing down your program by 10x for every single memory allocation. That means performance depends on how many allocations you do, which is really bad.

And just to illustrate what I mean by a leak. My definition is basically that you've got this routine, foo. And it mallocs some data. But it returns without freeing that data.

And nowhere else in the code log did you actually have a free. That's an actual memory leak. An accumulation would be if you have some free somewhere. But it's never actually called.

So a lot of the code that we write happened-- at least in the past, it's changing-- but a lot of it happened to be written in C++. And all of you, I guess assume, you know about new and delete and all those kinds of things that we do in C++. And it turns out, that those things are notoriously bad for assigning memory leaks.

So instead, what people typically do is they use reference counted objects. So every time you use something you increment some reference count. And every time you delete it, you decrement a reference count. And when the decremented reference count is zero it automatically gets deleted. Now of course, this only solves an actual leak. It doesn't solve an accumulation.

So here was the problem that I noticed. That was all kind of setup, just saying what's a memory leak, what's an accumulation, why is memory a problem. Let's get to actually why it was a problem.

I had this issue. I have our little server application. And it's running out of memory after several hours. Now I used Purify on a much, much smaller setup. It was about 1/100 the size of the actual setup, because larger than that, Purify would crash, because it would run out of memory. Because by the way, to track all the memory references, it's got to allocate memory, which ends up being a nightmare of unprecedented proportions.

So the deal was, was that I couldn't really detect this leak very easily. So what I did was I examined the memory that was in use. Essentially, what I'm trying to say is that, there wasn't so much a leak as there was an accumulation. Accumulation again being a logical problem. Well, yeah, all of my code is written correctly. But it's just that free call is never called.

So I examined the memory in use and I localized it to one operation that whenever we did it, memory would increase. So since doing it only once, there's a lot of

noise. What I did was I said, well, OK, why don't I just do this operation hundreds of times at the same time. Because then, if I see some allocation that's 100x from before I started this I can know that this was actually a problem.

And that's exactly what I did. I had this issue where something was being allocated 64 bytes. Now 64 bytes is so small that it's pretty much in the noise. But if you do it a hundred times, and you're seeing 6,400 bytes. Then you do it a thousand times and you see 64 times 1,000. Then it becomes something easily that you can then track.

Once I noticed that this particular piece of data was being allocated like this, I went through and I realized that this actually was-- it was an accumulation and not a leak. So someone would allocate something whenever an incoming message came in. And there was a free somewhere, but it was never being called.

So the lesson that I learned from this is that in a lot of situations with performance debugging, it's very helpful to try to find a problem-- in order to try to find a problem, it's very helpful to magnify it and make it a lot worse. If you think something is problematic in the network, slow down your network if that's an option. If you think something has a problem with memory usage, either do something like what I've described here. Or maybe reduce the amount of memory.

When I was in graduate school, we were working on a cache coherence protocol. And I'm going to spare you a lot of the details. But here's the thing. We basically stored a linked list for every data in memory. And we had to make sure that that linked list was rock solid no matter what the size of the inventory was. And if you exceeded the linked list, you'd store stuff in disk. So what we did to magnify any possible bugs was we created it so that you could make a linked list of at most one item.

That way no matter what you did it would force you to overflow and go into disk. And that was a great way to test a lot of overflow conditions that in practice would never have occurred. Because we do the engineering thing, figure out the capacity, double it, and hope that nothing happens. So it's good to exaggerate

edge conditions.

So that's one memory analysis problem. Here's another one. Here's the deal. You've got a user and they're using this middleware server. And they complain that with time it gets slower, and slower, and slower. And if you look at all of the-- if you look at CPU network and disk, you see that nothing is being saturated there. But if you look at memory, it's increasing pretty dramatically. And at some point, it increases so dramatically that the server application crashes.

So first thing, let's actually look at the memory utilization and see the timeline. What I show here is I show versus time, I show a chart of something in Windows that's called private bytes. Essentially, think of that as memory. What you see is that memory is growing slowly, whatever. But at some point, near the very end, near when the server crashes, it starts growing at a really, really fast clip. Now remember, I call this private bytes. What that essentially is, in this particular case, it's Windows giving a process some amount of memory.

It's not necessarily what the process or what the application is requesting. The application will request some amount. And Windows will give it this much. Well, Windows keeps giving it more and more and more. And it eventually just runs out of memory to give.

So the reason I made the distinction between something that an application and something Windows is giving it is that I, as an application, might be asking for four bytes at a time. Four bytes, four bytes, whatever. But what if Windows has this problem where, whenever you allocate-- it doesn't, but suppose it does-- suppose that every time you ask for four bytes, it says, well, you know what, I'm just going to give you 16 bytes because that's my smallest increment of memory.

You can imagine if I keep asking for 4 and it's dumb enough to give me 16 every time, we're going to see a huge expansion of memory, even though I, as an application, didn't ask for that much. So it's important to keep these distinctions in mind.

So the point is what was happening in this case, was my server's going fine, fine, and fine, until near the very end where it starts increasing in memory use at such a dramatic rate, that it eventually crashes.

So the first thing I did-- We just talked a lot about these reference counted objects. The fact that I keep track of whenever something is allocated and whenever something is not being looked at anymore. When it's not being looked at anymore, I go ahead and remove it.

If you look at the reference counted objects, they were pretty flat until you get to the very end of the run, exactly the same as when the memory was increasing. And it turns out, I've highlighted a few of these, and they ended up being-- it doesn't matter what they specifically were-- but in this case, they ended up being related to threads, and mutexes, and stuff associated with keeping track of what was going on in the system.

So some thread-related objects were increasing for some reason. Now turns out thread state in some sense is very cheap. So just because these guys were increasing, it doesn't really explain why we're dramatically increasing in memory. But I'll tell you what does explain it. If you take a look at a different attribute, which is these are the total number of threads in our system, we have a static pool of threads. And whenever a work item comes in, we allocate a thread. Whenever that work item is done, we deallocate the thread. So we keep it to a minimal number.

And it turns out, we had a cap. We said, OK, I'm going to allow at most 20 operations to occur in flight. Anything beyond 20, I'm going to queue it up in memory. And we'll deal with it later.

Here's the deal. What you see when you look at the number of threads being used is that-- it's the bottom chart, unfortunately, it's a little hard to see. But the point is it goes up and down, up and down, up and down with time. So stuff gets allocated. Stuff gets used. Stuff gets removed.

But then when you get to the very end, you see this monotonic increase in the

number of threads being used. It never goes down. And at some point, you hit the point where all of the threads that you've preallocated are completely taken. You obviously can't allocate any more threads. So every incoming request, you go ahead and queue.

This was the root of the problem. The root of the problem is that in this particular situation, each of the times that it was increasing was because there was again an uncaught exception. That uncaught exception resulted in a thread not being deallocated. At some point, we ran out of threads. And every single request that came in was being allocated in memory and waiting around for a thread to become available before it could be finished.

Well, if no threads are available, these things can never get pulled off the queue. You see a monotonic increase in the amount of memory. And you eventually die.

And so, it's actually a correctness issue. Why are you not catching exceptions properly? And why are you not recovering properly from them. But interestingly, it manifested itself in a performance issue.

So in each of these cases, what we've done, is we've looked at customized profiler instead of a tool that we might have gotten, Purifier, Valgrind, or whatever-- it's nice that it's tailored to our application. And we can make it as arbitrarily fast, because it understands what our application is doing. And it also-- the other nice thing about this is this can be run in production environments. If you write your own tool, and a customer calls and complains, you can just say, oh, just twiddle this knob and you're done.

That's kind of convenient. But of course, the disadvantage is that now you have to-- whenever you rewrite your code or do a new rev, or whatever, you have to make sure that these tools work right. And you have to recompile the code. If you have something like Purifier, Quanitify, in a lot of situations, you can run with some sort of preexisting binary-- sometimes, not always. You can run it without having to recompile. And that can be very convenient.

So I don't know. I was kind of stuck for what a lesson this was besides common sense is useful. So I just said memory profiling is pretty critical. Don't ignore that when you're writing your applications. And I think I also want to point out that-- I didn't put this here. But you might think that when you're using a garbage collected language, like something like Java, that these problems go away. But they actually don't. The problem is actually now memory allocations and deallocations are more hidden from you. So you have to be a little bit more careful about how you analyze them.

Case study number seven. So we've looked a lot of things related to CPU usage. We've looked at a lot of things related to memory. Now let's look a little bit at the network. Here was the problem. I have a user, our Canonical admin. The user wants to perform an operation on some virtual machine somewhere. Remember the flow. Issue request, it goes to some server. The server goes to the host where that VM is. Does some operation, comes back.

So this user wants to perform an operation on a virtual machine. I had setup A, In setup A, this operation end to end took about eight seconds. Fine. That doesn't mean much until you compare to setup B, where it took twice the amount of time. It took 16 seconds, not 8 seconds. So now the question you're asking yourself is what's the difference between setups A and B.

In setups A and B, the only difference is that this middle server is a different version number. But everything else is exactly the same. So with this different version you somehow manage to make your performance twice as bad, which is a really bad thing. We got to get to the bottom of that. And in addition, it's not like it was some random thing that happened once. It happened every single time.

So now, let's take a look at what was going on. So again, a lot of times, I tend to do things in a top down manner, where I say what's the easiest information to collect. And then, OK. What's more difficult, what's more difficult?

And finally, if you have to use really industrial strength tools, you use them. But it might not be necessary. In this case, my first thing was just to use logging and try

to figure out where time was being spent.

Remember, we've got a client talking to a server, talking to an end host. In this case, the latency imposed by that client was exactly the same in these two setups. So that's not where this eight seconds is being lost. The amount of time being claimed by the server was exactly the same.

I'm a client. I go to a server. It then goes to a host. However, the time spent on the host was different. That's where all of the time was being spent.

Now, you're probably looking at-- well, you may not be. But I'll tell you to look now. We had setup A. Setup A was a client and the server were each at some version number. And the host was at the same version number. In setup B, the client and the server were at a different version number with respect to the host. But we just said all of the time is being spent in the host. So why is that going on? And this is where understanding the architecture is important.

It turns out in this case, the first thing that happens when you have a difference in version number is that the server helpfully talks to the end host and says you know what, you're in a different version number. So let me give you some code to run, because you're in a different version number. So in fact, even though the host is at the same version number, there's a small shim running on that host that makes up for the fact that it's not at the same version number.

So now let's analyze this host and figure out where is this time being spent.

I did a little bit more logging on the host. The reason I did this, because a lot of the standard tools that you might use, like gprof or whatever, these things didn't exist. We have our own kernel. Which therefore, we have our own tools and our own nightmares, and headaches associated with it.

So we had to use these tools instead of commercially available tools. When you use these tools, we ended up narrowing down the time. Here's what goes on. I'm a client. And I talk to the server. The server talks to an agent on the host. It says, agent, this is what I want to do. The agent then talks to the hardware abstraction

layer and says, dude, this is what I want to do. The hardware abstraction layer comes back to the agent and says, done. Agent goes back to the server. Sever goes back to the client, blah, blah, blah.

It turns out that this interaction on the host between the agent that accepted the call from the server and the hardware abstraction layer, that was where all of the time was being spent. It was 10 milliseconds in one case. And 20 times that length in the other case. And so this was a pretty big difference not to be ignored. So we wanted to look at why that was going on.

So here, it's kind of a dumb thing. But the next the thing I did was, well, let's take a look at the configuration. Maybe I did something stupid that would have caused this to happen. Here's the deal. In setup A, remember everything is at the same version number. And it turns out this agent HAL communication occurs over a named pipe. OK, whatever.

It turns out in setup B-- remember the client and the server were at a different version from the host, so the server has to download some shim code onto that host so that everybody can talk correctly. It turns out that shim code was communicating with the hardware abstraction layer-- instead of over a named pipe-- it was communicating over TCP/IP.

Now I have to give credit to my colleague on this. As soon as I showed him this information, he hit a flash of inspiration. It turns out that when you're using the named pipe and it takes 10 milliseconds, the reason that's different from TCP/IP communication is because of something called Nagle's algorithm. Nagle's algorithm basically says, you want to accumulate data before you send it so you reduce the number of round trips and the overhead of creating a connection, and whatever.

And the problem was this was biting us. Essentially, we were waiting within that shim to collect enough data to send it back to that HAL. And we were waiting fairly needlessly, because what happens is that this algorithm waits for a certain amount of time. And then if it hasn't gotten more data, it goes ahead and sends it. So we

are basically stuck, waiting for this time out to occur before the data actually got transferred.

And there's a very simple way to get rid of that. You just tell it, hey, don't use Nagle's algorithm. We don't care. And that solves the entire problem. So the point is once you get rid of that, the performance is exactly the same.

It's a very simple configuration setting. But that's again, a situation where somebody-- and in fact, I remember asking why this particular change was made. And again, it was, well, we didn't think it was going to be a big deal. And that ends up being, again, a very bad way of doing performance engineering.

So definitely, the lesson that I learned here is just like the lesson with 32-bit, 64-bit. You might think that a little change like this doesn't mean anything. But it really does. And here, it's actually important to understand the entire stack end to end. Because otherwise, you're never going to be able to figure out where the problem actually was occurring.

I already alluded to a previous situation where we had a correctness problem, which masked as a performance problem. In the previous case, it was because I had a limited thread pool. And whenever I ran out of threads, I'd queue everyting. And at some point, I had a correctness problem where I kept killing threads and never resurrecting them. And so I ended up queuing everything. And everything went to pot.

Now let me talk about a different situation. Here I've got a poor little customer, in fact that customer was me. I was powering on a virtual machine. Remember it goes to the server, goes to host, powers on the virtual machine, tells me it's done. Every other time I would do this, it took five seconds, which is pretty OK.

In comparison, every other time I did it, it would take about five minutes. That's a pretty huge expansion. And if a customer called me and said this, I would basically be in deep doggy do-do.

So it was important to try to figure what was actually going on here. So why does

powering on a VM have such a variable performance? Well, the key here, again, is to understand what's the end to end path for things.

In this case, what I know is that when you power on a virtual machine, one of the things you have to do is allocate some space on disk to write swap information and whatever. So one of the first places I looked was I said, well, let me understand how the disk is performing in each of these cases to try to figure out what's going on. It's a mixture of common sense and knowing what's going on, and then knowing where to look.

So let me let you in on a little secret. In the real world, if you're using some sort of disk, if your disk latency is basically 10 milliseconds or less, you're more or less golden. That's not a problem. If your latency is between 10 and 20 milliseconds, maybe that's acceptable because you have a complicated topology. You're some multinational corporation with sites everywhere, whatever.

If your latency is getting to the point where they're 50 milliseconds, then you probably have to start worrying. That's actually quite a bit high. If your latency is greater than 51 seconds, that is staggeringly bad. In this chart, I show you the latency versus time. And what this chart shows you is that every single time step, it says in the last five minutes, what was the highest disk latency that I saw.

And remember our rules of thumb. 10 milliseconds, good. 20 milliseconds, not so good. 50 milliseconds, not so good. This one is 1,100 milliseconds. That's like you could walk to the disk, get the data, and bring it back faster than what it's actually doing. That's when you've got to call a priest. OK. That's horrible. So this was the reason that five seconds versus five minutes.

So now the question is why is this going on? Why am I seeing such unbelievably bad disk latency? And remember, the title of this case study was "Correctness Impacts Performance." So let's take a look. What I do here is I take a look at all of the events that are happening with respect to that disk. You're not going to understand this. It's some junk related to our product. But I'm going to circle the

relevant portion.

You'll notice a little line that says lost access to volume. Here's the deal. I send a request to the disk. It turns out the disk is connected over some channel to some channel, a disk controller, which is talking to some disk. It turns out, I kept losing access to that disk. Whenever I would lose access, it would keep retrying. And that's why ultimately, yes, it would complete. But it would take over a second per disk request. In the situations where I was not seeing that loss of connectivity, things were occurring at the order of milliseconds and my power-on would take five seconds.

So the point here is that in this case, it was actually a correctness problem with my controller. And the controller would basically flake out and not talk to the disk properly. And this manifested itself in a performance problem. Once I changed the disk controller, performance problems went away.

So if you're doing this kind of debugging, sometimes you have to eliminate-- I guess it's a Sherlock Holmesism-- that you eliminate the obvious and whatever's left must be it. A good example of this is my ex-officemate, he used to ask this question whenever he'd interview somebody for VMware. He'd say, OK, you have a networking problem. Where is the first place you look?

And everybody that's coming in with a grad degree or whatever, they're like, I would look at the stack. I'd look at the CPU. I'd look at a cache counters, or whatever. And my officemate would just look at them and say, why don't you just check the cable and see if it's plugged in. And surprisingly, it's so dumb. But when you're talking to customers, they're like I swear I did that. I swear I did that. Their swearing doesn't necessarily mean anything, unless it's directed at you.

So that's case study nine-- eight. The last two case studies are going to be specific to virtualization. And so what I'm going to do is talk a little bit about how the CPU scheduler works in the VMware server class product. And then I'll talk about the last two case studies.

So actually how many of you have used VMware before? Just as a hand. Thankfully, Saman has used it. That's good. So basically, the idea is that you're multiplexing time on the CPU between different virtual machines that are running on the CPU.

So what I show here is I show ESX, which is our server class product. Basically, think of that as the hardware. So you're running virtual machines on hardware. That hardware has, in this particular case, four physical CPUs. No problem. You've got four physical CPUs.

And you want to run virtual machines on this hardware. So you've got one virtual machine. Well, that's fine. There's four CPUs. One virtual machine. No problem. Plenty of CPU to go around. Now you got another one. Still no problem. They're not chained to any given CPU. And there's plenty of CPU to go around. No problem.

Same with three and four. You got all these four virtual machines. It's not a one to one correspondence in terms of CPU. We don't work that way. But there's plenty-- the point is, there's plenty of CPU to go around. Now what happens, and when VMs are basically available and when there's hardware on which they can run and they're using the hardware, that's called the run state. So running means I'm golden. I'm using the hardware.

Now, let's add another VM. I put it in red, because this poor VM, he wants to run. But you'll see that every single CPU is actually taken up by another virtual machine. So even if this virtual machine wants to run, it can't. It has to wait. The time that it spends ready to use the CPU, but unable to use the CPU, because there's no CPU available, that's called ready time. In some sense, it's kind of the demand. Ready plus signals down.

But anyway, the point is it's sitting around waiting to use the CPU. But it can't, because other people are using it. So it has to wait its turn. It waits its turn. And eventually a VM, our ESX scheduler is smart enough to realize that it has to fix the situation. So it deschedules a VM-- in this case, VM 1-- and goes ahead and

schedules that other virtual machine that was previously waiting.

Now predictably, the VM that's now ready to run but can't is in the ready state. And the VM that formerly was in the ready state is now using the CPU. So it's in the run state. It's very happy.

The final thing that I should point out is that VMs don't have to be in the ready state. They don't have to be either using the CPU or ready to use the CPU. Some VMs are voluntarily descheduled, because they have nothing to do. Some VMs are descheduled, because they're waiting-- they're basically waiting for an IO to complete. And so they don't need to use the CPU. And that's what I depict by VM6. These are in what's called the wait or idle states.

So hopefully, you now have a fairly rudimentary understanding of what our CPU scheduler is doing. Let's talk a little bit about a very simple performance problem related to this. And I call this-- But It's Only a small probe VM. I'll explain what this means in a minute.

Essentially, you've got two of these hosts, two ESX hosts that both want to run VMs. And I've got a user, depicted by the user icon. And he's talking to this one VM. Let's call this VM the vSphere VM. Please pardon the nomenclature. It doesn't really matter. But anyway, he's talking to this vSphere VM on one of the hosts.

Now, it turns out that that VM that he's talking to, as I depicted in the diagram, is talking to another virtual machine, which I have labeled vSphere database. So it turns out, this vSphere VM is talking to a database over what's called an ODBC connection. That doesn't matter. Don't worry about that. And it turns out that on that other host where the database virtual machine is running, we have another virtual machine. It's called the probe virtual machine.

The job of that probe virtual machine is to sniff any traffic going into the database. So as I've said, the vSphere VM communicates with the database. This probe VM is monitoring any traffic that's going to that database virtual machine. And therefore, as you can imagine, the more traffic that's being sent between this

vSphere VM and the database, the more work that that probe VM has to do.

Here was the complaint. My colleague called me up, and he said, hey, I'm having a little bit of a problem with my vSphere VM. All of a sudden, it's completely unresponsive. So then I was like, OK, well, this is kind of silly. Let's try to figure out what's going on.

So here's what I did. I graphed various metrics, which I just spoke about, the used time and the ready time, on this chart. The time is on the x-axis. And the y-axis is in milliseconds. In what I'm going to call the land before time, the ready time of the database. So look at the circle on the-- let me think, you're-- on the left. This is the ready time of that database virtual machine. Remember what we said, when you're accumulating ready time. It means you really want to use the CPU. But you can't because all the CPUs are already in use.

So it turns out that the first thing that happened when my colleague called and said I'm having a problem, is I look at this chart, which is versus time. And I said, let me guess. Are you having a problem at about such and such time, which corresponds to the middle of this chart? And he said, yeah, that's when I'm having a problem. So you can tell that there are two very clearly demarcated regions here.

In the left, you can see that this ready time, it's around 12%. Now if you're like me, a number that has no context means nothing. So 12% means nothing. However, what I'd like to point out is that when things got really bad, that number spiked up to 20%.

So here's the deal. I'm a database virtual machine. I want to use the physical hardware. I can use it about 90% of the time, because my ready time's about 12%, whatever. But at some point, something happens where I now can only use it 80% of the time. And it turns out that that difference was enough to cause this vSphere VM talking to this database to be perturbed. And that ultimately was the customer's problem. So the question is why. And here is the reason.

Turns out, here's the deal. Let's go back to the diagram so I can explain this.

Remember that the vSphere VM talks to the database VM. Remember as well, that the more traffic there is, the more work the probe VM has to do. It turns out that what this user did was at that critical juncture, he started a workload. In starting that workload, he caused a lot of traffic between the vSphere VM and the database, which therefore caused a lot of work by this probe VM.

Well, what's happening? The probe VM and the database VM are sharing the same underlying CPUs. So if one of them is running, the other one can't. And that's why we saw this gentle increase in the amount of ready time to vSphere database.

Well, this has a cascading effect, because it can't get as much time to use a CPU. Any request that goes to it gets slowed down. Where are the requests coming from? They're coming from this vSphere VM. Who's initiating those requests? It's this poor admin.

The bottom line is that this admin is seeing poor responsiveness because some doofus put his probe VM on some other host, because he wanted to see what was going on in the database. In fairness, I was that doofus. But that's fine. We needed that, because we had to debug other performance problems.

So the point is that one of the things that's kind of important in this particular case is that whenever you introduce any sort of monitoring-- it's like a Heisenberg thing-- it's going to have a performance impact. And understanding that performance impact is pretty key. It might even shift where the bottlenecks are in your system. If your thing has a high cost in terms of CPU, it might shift the overhead of whatever's running to the CPU where it wasn't there before. So it's kind of important to understand what you're monitoring and how you're monitoring it.

OK. We're almost done. This was a really curious problem. We have a customer. And a customer was performing a load test on a server. And in that load test, what they would do is they would just keep attaching clients to that end server and run some test.

At some point, they would attach clients and their workload suffered, their workload performance suffered even though there was no metric that was saturated. And so the question was why. Why you keep adding clients, nothing is saturated, but performance suffers?

Well, this chart's kind of an eyesore. But let me explain it. The first thing we do is we've got some profiling information. On this chart, I show two things. In purple, I show CPU usage. And in white, I show disk latency.

As you can see from this chart-- and the y-axis is just 100%. And they're both normalized. As you can see at some point, the CPU usage is steadily, steadily increasing. And at some point, the disk latency takes over and gets to be really large. You can tell that, because up until about 3/4 of the way in, the disk latency hovers around zero. And all of a sudden, it jumps much higher.

So my first instinct was to tell the customer you've got a disk problem. Your disk latencies are going over a cliff. And that's why you're having a performance problem. Unfortunately, that's not correct. Why is that not correct?

It's not correct, because, yeah, disk latency gets worse at 4:00 PM. It's because of swapping and some other nonsense. We know that. However, the application latency actually gets worse at 3:30, not at 4:00. It's half an hour earlier.

So now the question is what's going on that causes the problem, even before disk kicks in and gets to be another problem. This is where understanding the difference in metrics becomes pretty critical. Now, I'm not expecting very many people to understand this chart. But I will try to explain it.

What I do here is I show a different chart, which essentially shows CPU usage of all the different virtual machines and processes on the underlying host. Forget about what's in most of this chart. And just focus on what I've circled.

I have two columns shown there. One column is called percent used. And one column is called percent run. You can think about it and think, OK, well, I can imagine used time means probably how much time I'm using the CPU. But then

what is the runtime? Isn't runtime the same thing, because I'm using the CPU? What's the difference?

This is actually the critical problem. It turns out, that when you talk of a percent used time, you're normalizing that to the base clock frequency. If I've got a three gigahertz machine, I've normalized it to three gigahertz. However, we all live in the era of power management. It turns out this percent run, which nobody understands, is normalized to the clock frequency at the time you were running. So if I did power management, and I was only running at two gigahertz for a certain amount of time, percent run captures that.

And so in this particular chart what I'm showing is that a bunch of virtual machines are seeing used time different from runtime. And the runtime, which is normalized to the clock frequency of the time you're using it, is actually greater than used time, which suggests that the user's actually using power management. And it turns out, that power management was kicking in at some point and causing their application latency to go down even though the CPU was never saturated.

And in this case, when we went back to the customer-- you look at this chart, and you instantly see, OK, I know that because this is normalized and this is normalized to that. This is what's going on, you have to first of all change this parameter. They changed it. And they were unbelievably happy. It was pretty cool.

Now, I have to admit that there's much more to this story that I'm not showing you. Which I'm not allowed to show you. But it becomes a very interesting thing. But the point is it's very, very important to understand your metrics. Here just understanding the difference between runtime and used time made all of the difference in diagnosing this customer's performance related issue.

So I just pick a smattering of ten random bugs. And in each of them, I tried to distill one kind of conclusion you can get from that particular thing. Because all of you can read, I'm not going to bother repeating all of these things. But anyway, here's the first five. And you all get the slides. And here's the second five. Oops. I'll just keep that up for a second.

And in order to conclude this talk, let me just say that it's very important to avoid assumptions. It's very tempting to say well that's-- I'll give you a great example. I used to interview people. And I'd say, OK, you've got a UI. You've got a problem. You have a UI. And you click on something. And it takes a long time. Why is it taking a long time?

And the first thing they'd say, well, I'd go and look at-- this is literally, not joking. They'd say, well, I'd go to look at the cache messages on the server. And I just looked them and I said, how do you know the client's not the problem? Oh, the client's not a problem. And you base this on what knowledge? Don't use assumptions. It's a very bad idea.

It's really important to understand the entire system that you're dealing with. From the client to the server to the hardware to the network to the disk. You have to understand all of those different pieces, because you have to understand where to look. And what's really onerous is that-- it is not so much when it's a problem in your code, because you can be blamed for making stupid mistakes-- it's when it's problems with other people's code and you spend a month looking at other people's code, figuring out what's the problem That's when things are really kind of irritating. And especially when it's a problem with hardware.

And I think the final bullet I'll say is be persistent and be thorough. I don't have to tell you how to be good workers. So let's just blah, blah, blah. We're done.