So, in this talk, I'm going to be talking about emulating CXL with QEMU. And, if you don't know me, my name is Adam Manzanares, and I work at Samsung. And, I'm very focused on open-source communities. And, this is a work with many collaborators across many different companies, and I'll be more explicit about that at the end of the slides, but it's all about emulating CXL.

So the first thing I'd like to get people to know about, if you haven't heard about it, is QEMU. So what is QEMU? It's a generic and open-source machine emulator and virtualizer. So we got to unpack this, right? So you got to kind of take a step back and realize what this means, right? This is software that you can run on say Linux machines, and I believe it runs on Windows as well. There's, there's multiple platforms it runs on, but what it can do is it can actually emulate another architecture. Say you have an x86 laptop and you want to run an ARM-based system on there. One of the things that it can do is actually emulate the actual ARM CPUs, and then you can run ARM code on top of this QEMU emulator. In addition to emulating CPUs, also many peripherals are emulated as well, and this becomes very valuable when we talk about CXL, and I'll talk about some use cases that came before CXL to give people a sense of how valuable this has been. And you know, this is a storage developers conference, so I want to relate it to some concepts that might be more familiar with people from this community, and then we can kind of talk about some differences with CXL as we go.So yeah, like the first box what we see is the full system emulation. The second box that's pretty interesting too as well is you can have user mode emulation, so inside of Linux, you don't want to emulate the entire system, but you just want to emulate a program written to a separate instruction set architecture. You can run that from the command line and just execute a program for a different architecture. And then the last one is that there's a virtualization piece so it also you can leverage KVM or Xen, and so you can use hardware-assisted virtualization of a hardware platform, and then you can get these really fast VMs. And this is quite interesting because you can get this high-performance virtualizer coupled with emulated hardware, right? So some of the hardware might be slower, say for your peripherals, but the actual CPU is quite fast. And you know, I'll go, and I'm going to try to give a live demo at the end of this so people see what can take place. Yeah, and then you know, go check out the website QEMU.org and become familiar with this tool. It's very useful, and I'll also go into why it's valuable at a company like Samsung.

So this allows me to take a step back, and before we talk about CXL, right? So I am here at the Storage Developers Conference, and many people have been familiar with NVMe. And for NVMe, QEMU has been extremely valuable. And the way that I see this value is in: we can rapidly prototype hardware features inside a QEMU, and then we can write the system software on top of it, right? So QEMU is a very fast emulation platform, so you can run inside a QEMU and run your user space applications like near-native performance, right? There's some small deltas, but you know, it's for a developer perspective, it's extremely fast, and you can cut down your sort of like turnaround time. And developing your hardware and your software, so you know, you can really prototype end-to-end software for new features, and I call this working on the plumbing—this is the plumbing between the hardware and the applications. You can really have valuable prototypes, and I really want to encourage people to do this, especially if you're working in standards. Right? If you can go back to standards organizations and point out issues with say standards, and you know, we brought this up earlier, I was asking about what is the feedback loop for software people? I believe it's very valuable to have software developers work along with the standards people, and to end as well as hardware developers, and so you have these tight loops of feedback, giving information to each other, to come out with you know, hardware and software that's going to solve the problem at hand. So some of these examples, right? If you knew about data placement, so there's been a couple of different ways of approaching this, there's been the zone namespaces, there's been FTP beyond that, there's a technique to move data internally on the devices

called simple copy, there's ways of virtualizing the devices, and all of these hardware features exist in QEMU. We've had them in QEMU, in order for us to build the whole system software, and we can get a sense of how is software going to take advantage of the features? You know, we can't know the exact performance, like this is not the goal here, right? The goal is to be able to emulate all the hardware to get the software right, and then, yeah, you have like this really interesting complete picture of how an application is going to use it, and you can use this to inform your decision where you're going, and now I want to make it more specific to Samsung. So the larger team that I work on is called the Global Open Ecosystem team, and this team has people working in the Linux kernel, works in QEMU, works in user space software, and our core mission is to enable Samsung hardware, right? And it is pretty clear, at least in my mind, you know, looking at the space for the past 10 years, there has been more and more hardware software co-design happening, right? So we look at what is the end application? You know, are they going to how are they going to connect their application? What are the interfaces to it? There's many choices of how to do this, and which layers of plumbing need to get involved, and we've seen that it's important for us to understand these layers and potentially contribute right, in the end, we want to demonstrate the technology and work with partners, so that that's another big thing is that we work with partners, and being in open standards and in these open ecosystems and software allows us to do that. So you know, I like to talk about some of the successes that Samsung has had, and this is previous to when I when I've joined, but for the NVMe support, an individual who contributed quite a lot, his name is Klaus, and through his work, you know, he's done a lot of these features as well inside of QEMU, and we've also got a newer member who helps to review work in QEMU, so you know, the summary is that we still believe that this QEMU knowledge of how QEMU emulates NVMe drives is very useful, and we love it as system software developers, right? It is great. To have hardware, but to me, grabbing the hardware is kind of towards the final stages. If I can emulate this quickly and prototype it, I can move very fast and be ready for the hardware once it lands on my desk, and that is the true value, right? We're trying to get ahead of the hardware as much as possible and build our system software, improve it out, and then be ready for the hardware as soon as it lands on our desk. And what, what is this? What is the value here? Right? So, so we are a hardware manufacturer. We need to test these devices. We need to understand how customers may be using them, and all of this can move much faster before the hardware is available, right? That's the advantage that we're trying to come up with, and it's very tangible, and CXL in my opinion, and I'll get into more details into that, and another huge benefit that I see is that you just bring more people in right. The earlier you can share this code and talk about it and demonstrate end-to-end frameworks, you may catch somebody who's working on a similar problem and can leverage what you're doing. So that's another huge benefit that I see about this right? It brings people into the ecosystem, and again, you know, let me take a step back right? This was kind of setting the stage for why we would want to do this in CXL right? We we did this for NVMe, we saw tangible results, we continue to see valuable results from this, and so it seems to make sense to try to do this for CXL. CXL is very different, but QEMU NVMe emulation is kind of a good blueprint for what we wanted at Samsung.

So that leads me to my next slide, right? So again, I want to reproduce these success cases. There are clear advantages that I've seen when you understand the system software and you understand your hardware; you understand both sides of the equation, and you can build that end-to-end software. Without the hardware dependence, and then as soon as the hardware is in your hands, run your experiments, get the results, and have the know-how what to do with it. And as has been discussed earlier, so there are a couple of points I want to talk about from the earlier discussions, right? There is limited hardware availability. You know, to be completely honest, right, that's not something I face in the position that I'm at, right? Well, I'm I'm fortunate enough to be close to the hardware and do this for hardware. But software moves very fast. And you know, CXL is a new technology, so the hardware that is

available may be in limited amounts or you know, may not have all the right features at the right time, right? It is an emerging technology. Right, if anyone tells you otherwise, I don't really believe what they're saying, right? There it's it's coming. I see the hardware; it's there, and you can see it at demos and places like this, but it is still limited availability. And the two remarks that I wanted to make right, so previously there were two comments about what you need in your slides. One of them was you need some story around AI. I don't have that today, right? And then the second one is that if you you're talking about CXL, you also need the timeline, and I don't have a timeline in here, and I'll speak to that too, why I don't really see the need for the timeline for this work, even though it does influence our work, you know, I don't want to tell you that it's not a part of what I think about, but when I talk about the simulation, what's nice about this is we don't have to think about those timelines so strictly and give a concrete example. We can add CXL 3.X features as soon as the standards are available to our emulated systems and start testing out these features at least how the software would look. Right, we're not waiting for the CPUs to get the the the CXL standard version out at the time or the hardware to be there, and even though we know these things are coming right, we don't have to wait for them. And lastly, one thing that I'll point out, and I'll talk about this more as I go through this, talk it's two ecosystems right there's two ecosystems that can really benefit from this work that I've seen. One is the operating systems, right, that's what we normally talk about like we talk about the operating system and applications that are going to use it. We've talked a lot about this previously, but one of the ones that I've seen that has been super beneficial to as well, and we didn't initially start, you know, we knew this was coming, but didn't realize how fast it would hit us was management, like looking at BMCs and having sideband interfaces to talk to the devices right, is that this can all be done using QEMU and open source tooling right, so all everything's there for what you need to go and write your software right and improve out your software and be ready for once the hardware is available to you and mass production.

So now I'll get more into the details, right? So I was very confident when we first started doing this that it would be valuable because of the successes of NVMe. And I have seen the benefits already, and now let's get a little bit into which components are emulated. Before I get into that, I do want to remind people that I currently do not see the performance of like the QEMU emulated CXL memory to be useful for understanding your application behavior. Right, this is not something for you to go run and say, 'Hey, CXL will work well for me.' What it is is saying, 'Hey, will my software be ready to optimize for CXL software?' Right? So they talked about like placement using numactl with QEMU, you can see these NUMA nodes and place data on there, but we're not, we're not claiming that the performance will be measurable. And it's due to the implementation of how it's done so far that it happens to be slow because software is involved in memory accesses from the QEMU side. You know, there's been talk of alternatives, but largely to us working in this community that are developing this QEMU code going after the performance does not get the real benefit what we're looking for, right? It's it's just being able to get all the system software right. So now we'll get into a couple of CXL system components. And this is where I think we really need to think about what's different than NVMe, right? Because you need to start thinking more about all of these components that are involved. And it has implications like specifically I'll talk about one that has come up in the past couple months about virtualization of passing the hardware to virtual machines. Right, in NVMe it's pretty clean how to do this, right? But with CXL there's components along the way that it makes it much harder. And so the first component that gets emulated in QEMU is called a fixed memory window. So these are called CXL fixed memory windows. And what this does is from your physical memory space, it maps some of those addresses into a CXL host bridge, right? And so at this level, you have interleave and quality of service for throttling handled here. This is a great time to make the point that CXL memory access, .mem accesses has no software involved, right? You program these fixed memory windows, and when you read and write to these memory addresses, it's all through the hardware that's sending it to the devices, which is very different than

NVMe where you have to set up commands and send the device, right? This is all happening from the hardware as memory accesses, right? But you do need to set up these fixed memory windows ahead of time, and this responsibility could lie on the firmware or it could be the OS, and it depends on CXL versions, and there's detail in here but I don't want to cover that in here. So then below the this fixed memory window, you map towards host bridges, and it's very similar to PCI host bridges. And this is what has made working on this more tangible is that the fact that hey, there was a lot of this already emulated inside of QEMU so there were just some changes that had to be made here. And then from this fixed memory window, there's also an HDM decoder, which is a host-defined memory decoder. And this acts similar to the fixed memory window, but at a layer below, and then this maps from the root port down to the maps to the root ports under the host bridge, so you have a host bridge, and then a number of root ports. And then lastly, you know we've talked about CXL switches, and QEMU has had support for some simple CXL switches which is a single upstream port. It has an internal PCI bus and multiple downstream ports. This has all been emulated inside QEMU, and the thing to take away from here is this is configurable. You can set the amount of devices, amount of host bridges, right? You can set the fixed memory windows, you have a lot of control over this physical topology that you wouldn't normally have. And the last thing is for the memory devices, right? So we're emulating in QEMU, it's now emulating memory devices, which is a CXL Type 3 device.These are Type 3 memory devices. And it supports whether it's volatile or persistent. So this was baked into QEMU. Initially, I believe they were probably just persistent. But then we cleaned it up and made sure that volatile was added and that it was reusable in a reasonable way from the software perspective. So that's the components that you have inside of QEMU at the moment.

So now I want to kind of walk through this. So don't worry about looking at this text directly. Afterwards, if you go dig through here, you can grab the same information from QEMU's website, the CXL documentation. But I want to talk about what is here. So on the first figure, you have three fixed memory windows. And this was the top layer of what can be emulated in QEMU. And so there's just three examples here: either you go to the host bridge 0, or you go interleave between host bridge 0 and host bridge 1, or just host bridge 1. And this is at the top of this hierarchy on figure 1. And there's only one active at a time, so it's going to the first host bridge. The thing to take away from this slide, too, though, is that this is directly manipulating the physical address map. And this is set for you from the platform. This is the platform form where there could be only a set of options that are given to you, that you may not be able to change this. But in QEMU, you can play around with this. This is something that you can change if you want. And then the next level down, you're in the host bridge. Let's see if I can go here. So the first top layer was the fixed memory windows with three options. Then in the host bridge, you can see that it's just routing down to one of these root ports. And from this root port, there's a device attached here. And... So now this HDM decoder can route when the emulated CPU accesses one of these memory addresses, it gets routed properly all the way down to the CXL type device. And so that's one example. And then here, this one's a little bit different. And so you go to a single host bridge, and then there's a switch involved. So the only big difference here is that there would actually be a switch involved. And so the switch also has an HDM decoder on the upstream switch, which routes traffic to the downstream port. So it's very similar to the one on the left, except for the fact that you have a switch involved. And then the switch is going to have to have a HDM decoder mapping upstream ports to the downstream ports. But what is the takeaway, right? There's quite a bit of flexibility in looking at what topologies could be emulated here. And I want to kind of give the caveat too, right, is that this is not an effort that's done alone. It's done across multiple companies. And some companies have more interest in other parts of it. But we're all part of this greater ecosystem, and we need the devices, the switches, the host CPUs. All of this needs to work together. And so it has been pretty healthy, in my opinion, that we have buy-in from many of these different sources to work on QEMU.

So I want to show some high-level pictures here, and then I'll try to show a live demo and see how that goes. But this particular demo here, what it's doing is that one of our team members created a tool, and this is available. But what it basically does is it uses the open source ecosystem already now. And yeah, there's a lot of commands. There's lots of these different options. But it's wrapped around into a simple tool that allows you to, say, test CXL features. And so this one, it runs a QEMU system. And I'll show this in the demo. I think it's worth showing. Some of the run lines for QEMU can be quite large. There's many options for QEMU, and that can be confusing. And this is all hidden behind in this tool. And there's many ways of doing this, but I think for some people just wanting to try it, it might not be useful. And so then this tool loads all the drivers, all the kernel support. So the system is run. Then you actually connect to the system and put in the drivers. And then you can list the different devices. And one thing that has been done here, and I can talk about how bleeding edge this is, is that there's a feature in CXL called dynamic capacity regions, DC regions. And the intent is that a host would be able to release and free portions of CXL memory. And then this could potentially move to different hosts. But there's a framework for using dynamic capacity. And we saw this as an important feature. And we worked on the QEMU emulation and there were early kernel versions of the support. And then, you know, we kind of honed in on our QEMU, got that upstream. And then eventually the kernel community kind of settled on how they would do it. And we give input on this too as well, because we work hand in hand with the kernel side. And then you have like this complete end to end where this new set of features comes out and it's being tested directly on the QEMU support. And these are dynamic capacity devices. And so to me, this is a success story for the whole ecosystem as a whole, right? Is that we can work together and be ready to test when those patches are there.

And so, one thing that I wanted to point out that I didn't do in the last talk is that that tool was created by some developers on our team. Just to kind of hide some of these details, right, and you don't want to repeat the commands over and over again. And I think you can benefit from that. But to make it clear of how this all works, many of these tools come from Intel, it's called the ndctl. But at the core of what this tool is doing is it's talking to kernel interfaces, right? The sysfs interface, like it's setting properties of CXL, like you can see here. It's probably too small of the font, but it's /sys/bus/cxl. And then you have a way of communicating. It's communicating to these kernel drivers, and everyone has levels of abstraction, right? There's this ndctl framework, and then we have a framework on top of that, right? But our view is to leverage what works already out there, right? We don't want to write any software just because we can, right? We actually want to use the software that's available because we don't want to reuse. Our job is not to just write software for the sake of software.

So this goes back into the features. You can emulate. And you can probably get a hint as to why this is so valuable, right? And so you can emulate events. So CXL has events where it will say, you know, like a DC region has been created or some memory is now poisoned, right? There's a lot of commands that go on through what's called the .io channel. There's different protocols. Someone mentioned that in the presentations below. But you could think of the .io as sort of a .io as for management of like how to set up these HDM decoders, how to get events out, right? And QEMU has support for all of this. You can do firmware updates, right? And recently we just had like aborting background commands, you know, that we put that in QEMU. And then that's valuable for like some of these commands can last a long time and you may want to abort one. You know, like getting time steps, any of the logs that you identify, you know, sanitize, poison. Poison management, you know, you can look at more complex device types or something called multi-headed devices, dynamic capacity devices. And then there's switch. And then the switch has its own set of capabilities of what it has currently right now. And I'll tell you some things that I find very valuable, right? Is that we also have like sideband connection to that, right? Through QEMU you

can use MCTP-based sideband in QEMU. And then you can kind of look at management software as well. Yeah. And what I always like to point to people, right? Is that, so this QEMU work was initiated by Intel and the developer has moved to another company at the moment, but then it was really picked off, picked up by his name's Jonathan Cameron from Huawei. And, you know, he has run with this and he is, you know, kind of the gatekeeper of what goes into QEMU, right? He's the maintainer of this. And so bleeding edge support, you can always find inside of his branch. So if you're curious about what's happening and where the software is going, you don't go look at the upstream QEMU. You actually go look at Jonathan Cameron's and you can see where we're headed and what's going on and he queues up the patches that eventually go into the main, mainline QEMU. And so that's the place to go and always look for his latest branch minutes based upon dates.

So one thing that I like to show that, that has been there, right, is that, you know, with the flexibility of QEMU, it allows people to look at, you know, the dual mode, like you could have an NVMe device that exposes the LBA space over an HDM range and so has this load store accesses. And you know, it's been there, this has been there for a while. This was developed on 5.18, so it gives you kind of a sense of when it was worked on. But what is the key point of this slide, right? It's that the flexibility of QEMU allows people to look at ideas that they're, you know, kind of throwing around right now, right? And allows you to prove some value for it too, right? Like, there's a lot of stuff where that would actually leverage this, right? And there's much more beyond what you need to do to like really truly make this successful. But you know, there's a start here. There's a blueprint that you can see how people change QEMU to look at, you know, potentially making some hardware on the long term.

And you know, here's a couple of pointers that I have, right? And so this CXL test tool, I found it pretty well, and I'll try to use it for my demo, right? And... some of my friends use it to, and he wrote a blog post about how to test CXL emulation in QEMU. And so what does this guide meant for? It's meant for someone that has not used CXL at all to go and fire up one of these VMs and then poke around and see what it looks like, what are the, all the components involved. And if anyone is interested in these dynamic capacity devices, which in my opinion seem like the method moving forward of how to move memory around between devices, like within a CXL fabric, to me it looks like DCD is kind of the best way forward for many people. And so if you want to look at this feature and take a look at it, there's a test for these dynamic capacity devices. At Samsung, we do have a Discord server and we talk about open source emulation or if you have questions of anything here, you can join there and chat with us. But more importantly, Intel also, and I should have updated the slides on this, they have the Discord server where they have discussions about Linux kernel development on CXL and they share a lot of information there, and there's a nice history too as well. And so there are, how would I say this? I mean, there are multiple places where information is shared publicly, but I don't think everyone has all of this information. So feel free to ping me afterwards and I'll try to get you in the right place, and in contact with the right people.

So, a couple of acknowledgements. So I mentioned this work for QEMU was started by Ben Wadowski, and he was at Intel at the time, and now he's at Google. Jonathan Cameron has done a lot of the work on Huawei. Ira from Intel is heavily involved on the kernel side and on the QEMU side. Gregory Price, who's at Meta, but he did this work previously at another company where he did a lot of QEMU work and kernel work, and continues to work. So he's doing a lot of work on it. Fan works with us on this, and Samsung, and David Lower as well at Samsung. And then there's Tong at Samsung. And there are many others, right? I can't add the names of everybody that's worked on QEMU, but I'm very thankful that people are starting, are looking at it, working on it, and helping us move this ecosystem forward relatively quickly in my opinion, right? We're pretty good shape for where the hardware's at.

So let me try to do a demo here. And then I'll open it up for questions. All right, so what I'm going to do, right, and so I think this is fun right here, right? One thing that I'll kind of point out now. If anyone's been using Windows lately and you've tried this Windows subsystem for Linux, it's quite interesting, right? You can run full-blown Linux on Windows machines and it also has support for nested versions and virtualization. So what does that mean? So this Windows subsystem for Linux is running as a virtual machine, but it supports hardware acceleration. So if you launch a virtual machine inside of this Windows subsystem for Linux, it runs at pretty much the same speed as if you were running on the hardware. So it's quite interesting. So I'll start one of these machines. All right. And so I'll kind of walk through. Maybe I can make it a little bit bigger for people. So this points out a little bit of debug information. So what is this tool doing, right? So it looks to see where I have a kernel. It looks to see where I have QEMU, right? This is like where the source code lies. And this tool is nice because it will build a kernel for you. It will build QEMU for you that you can use. And then it also helps you build an image because you need some like disk image that you pass to QEMU such that it runs. And so it just gives you some of these different options like an KVM is the hardware acceleration that I can use. And so this thing has started running. And yeah, there's some problem here. I don't know why it's been giving me this problem about it's missing some file it's looking for. But the most important thing is that you can access this machine. And so I'll send a command to it to look at the memory. So this is what's interesting, right? So now in this virtual machine there's eight gigs of memory. And it's split up into these two separate ranges. And then if I look at the CXL topology it's very simple, right? There's nothing really showing up in there because I haven't online the devices and set things up, right? But what I can do is pass --cxl to this and then I'll walk through what it does. So first it's going to install these modules. So CXL requires the module support. And I'll go back and go through it. And then it's slowing down now because it's trying to initialize this region. And as I previously alluded the performance of the CXL memory is not what anyone's chasing, right? And it's just the way it was implemented it's more similar to like PCIe, MMIO, right? And it's just a legacy artifact because that's good enough for what we need. And so I'll go back through this once this finishes. So I can actually go through it now. So what is this tool doing? So one thing that it's trying to do and I mentioned this before this tool relies heavily on ndctl. And there's a legacy there of that CXL memory even though it's volatile because it potentially can be added after the OS runs and potentially removed which is a much more controversial topic is that the tool for dealing with persistent memory is very useful in that regard, right? It's because persistent memory could kind of be isolated and memory that comes and goes from an OS perspective probably makes a lot of sense to isolate. So we check to see the ndctl is installed you skip installing it but then we go and we mod probe all of these modules that are related to CXL memory and then now if you look here that we do CXL list inside of this virtual machine you can see two of these mem devs there's mem 0 and mem 1 so they're both 512 megabytes but we have two but then you create a region over one of these mem devs so you're going to route some traffic down there and this is with the HDM decoders you end up doing this and the key thing here is software is involved in this and one thing that can be confusing for people especially when I go back and talk with people is that in early hardware there is some platform firmware that deals with these devices as well. So, some of the responsibility could be the OS—could not be, but be careful with this as you're moving forward. And so, this emulation is like CXL 2.0 where it assumes that the OS is going to be doing much of this, even though that it's attached while the system is booted, the QEMU is not getting involved in that, so the OS has to do this. So, then you create another region, you create a namespace over it, and then one thing that we can do is configure this DAX device, make it into system RAM, and then if you see here, there's a new memory block added, and it's small; it's only 384 MB, and some of the capacity got used for metadata. I think there are 128 MB memory blocks as you see down here, so some of it is done for the struct page, and it's just because we did 512 MB of memory. If you had a larger device, the 128 MB, whatever you need for your metadata. But as you can see, this same system now has some extra memory available for people to use. Generally, with these

tools, and so the key takeaway is all of this works now. There's lots of different tooling for people to go try out. Not all of it is documented well. I can be completely honest about that, but I think the people involved are pretty willing to try to help bring others along, and so reach out to us and work with us. I think that's how I'll end it on there, but it's here, it works, and we're willing to work with others who may be interested in this space. Thank you.

Any questions? Grant? I'll go, Grant.

Why won't QEMU emulate Type 2 devices?

Yeah, so, can I repeat the question? So the question is: Why is QEMU not emulating Type 2 devices? So let me make that clear on the background: Type 3 devices are seen as memory expanders, and Type 2 devices have the additional condition of having the dot cache protocol and can be seen as potentially having some accelerator function. It's kind of like a high-level overview of what you'd say of Type 2 devices, but if I take a step back, I'm biased—flat out, I'm biased. Why do I do this open ecosystem work? It's to support hardware that I see in the pipeline. Where I'm at, the most tangible thing I see right now is a Type 3 device. I know of other companies working on Type 2 devices through public mailing lists, so I think AMD has been very clear that they have some smart nick that has Type 2 support. I would encourage them to kind of look at this. In my opinion, those who have the hardware stand to benefit by emulating—not every exact feature, but it can help the ecosystem—and so I don't see that strong need from my standpoint, but I definitely think that would benefit the overall ecosystem. So go after those people that are publicly talking about their Type 2 devices, right? They're there, it's all public, okay, Andy.

The question is: If we look at DCD as a way of moving memory, how does that compare to the SDXI initiative, which tries to make a standardized interface for moving memory between locations?

Okay, for me, DCD is about the memory being allocated on a larger granularity, like I would say, gigabytes level, for a given host. So, say host A needs 16 gigabytes of memory now because they're trying to pack more VMs, and this happened at the scheduling process. So, to me, the granularity is very different in the timing of when you would want to move memory from, like, an appliance into a host system. But it's at that granularity, like huge, much larger chunks of memory would be moved through DCD. And the reason I say this is that DCD is a bit involved; you have to create this region; you may have to tear one down; you have to look for any memory that might be using it, so it's a complex process. Whereas, what I understand about SDXI, it's trying to standardize more of like DMA, like data movement. The memory is already allocated to this host, but the host is aware of multiple types of memory, and you just want to move small amounts of memory, or potentially large—I don't think that matters as much—but it's to support some function. Whereas DCD is fundamentally about just allowing a host to have more memory, like if you wanted to give more DIMMs magically, some way of thinking that way, which I think is very different than the goal of SDXI.

I have another question. Just a correction: SDXI can move large amounts of memory, but you're absolutely correct; it needs to have the memory allocated to it.

Yeah, yeah, I'll repeat the correction: it's that SDXI can move lots of memory around, but the memory has to be attached to the system for it to move around. The system has to be aware of it somehow, and DCD is a mechanism that gives systems memory...

Can you mention the idea of passing CXL memory to a VM? If you talk about VMware VMs, you can already assign particular NUMA nodes to a VM.

Okay, let me repeat the question, and this is an important point, so the comment was that if you're looking at VMware today with CXL memory, that it can be aware that certain NUMA nodes can be passed to a guest. So, here's the comment, which is the tricky part: this all works for us now, so people have talked about for type 3 as long as the host system sees it, and the guest is comfortable not caring that it's a CXL device, this all works because you can pass memory ranges, no problem. The problem happens what happens when something goes wrong through these .io interfaces, and there's poison, and you have to say the hypervisor is the only one who has that knowledge and has to use that knowledge, or do we say, 'I can pass the entire device up to a guest who's CXL aware?' That's undetermined, and the hard part would be passing the entire device because, as I talked about, decoders along the way, right, in the CXL fixed memory window, then the guest needs to be aware of all these things as well, right? So, that's the very tricky part about this.

And my third question is, you said you have no...

So, the third question was: I don't have a shortage of CXL hardware, and Andy would like for me to share, so I can speak to that. There are initiatives at Samsung—one is the Samsung Memory Research Center—and the plan is to put CXL hardware in there. The high-level plan is, and they've shown some of their results with this related for partners. So, I know someone here who's involved in this, and I'll introduce you so you can follow up with them.

Are there any other questions? Alright, thank you everyone.