CXL and SmartNICs: a paradigm change?

Alejandro Lucero Palau
alucero@os3sl.com

Abstract

Efficiently sharing memory between CPUs and high performance devices like GPUs is what CXL tries to achieve. This new approach will bring changes to how operating systems do things, and networking will also be affected sooner or later. Although similar functionality can be achieved with adhoc vendor pathways, CXL will allow simpler hardware designs, a protocol more suitable than PCIe for memory load/store operations by the CPUs, coherency managed by the protocol, and, what this paper states, the opportunity for standardizing how the control path is programmed by the Host into SmartNICs with Match and Action Tables. If the Linux way of supporting such control path, that is a slow path through the kernel and a fast path through the SmartNIC, is assumed to be the right solution, an CXL-based design could overcome the limitations of current kernel approach with TC and netfilter/conntrack when used in massive virtualization scenarios. Moreover, the offloading of rules and flows could not only be standardized but simpler and more efficiently done through CPUs memory operations instead of per-vendor driver code requiring helpers like kworkers or suffering hard-to-parallelize TC functionality. A paradigm change like this could open new possibilities bringing closer the dream of full, private and compartmentalized programmability in the network control path required in multi-tenant cloud networks.

Keywords

CXL, SmartNICs, MATs (Match and Action Tables), slow path, control path offload, OpenFlow, Clouds Multitenancy.

1. Introduction

New technologies can always be disruptive but only if achieving the expectations once the marketing dust settles down and the advantages can clearly be seen without the blurry vision of just-to-prove promises.

Memory is one of the most disruptive in potential when the term includes volatile and non-volatile options, with the industry looking for the Holy Grail of a high performance, low latency, low energy and non-volatile solution. The disruption would not only be in the hardware world but also in operating systems design since the current memory hierarchy would need a revision in terms of why operating systems do things as they do now.

While the memory-type innovations, so promising some years ago, have not achieved the expected status, except for some minor options for embedded systems, and, of course, the use of SSDs in servers, the memory connectivity revolution could bring such a disruption sooner, and CXL seems to be the standard gaining momentum (without discarding other options worth to consider and equivalent in the functionality envisioned).

CXL **[1]** can be seen as an PCIe extension. In fact, the current PCIe protocol is available with the CXL.io specification, one of the CXL protocols. The other two are CXL.mem and CXL.cache and the ones to focus here. The interesting thing about CXL is the handling of memory coherency between Host and Device, along with a more suitable protocol for the bandwidth and latency demanded by load/stores when performed by the CPUs. Depending on the requirements three different device types are defined:

1. Type 1 device using CXL.io and CXL.cache, the idea being the device coherently caching Host memory. The device can have a memory, apart from that dedicated to the cache involved, but it is not managed by the Host.

2. Type 2 device using the three CXL protocols, the idea being a type 1 device plus device memory managed by the Host.

3. Type 3 device using CXL.io and CXL.mem, the idea being memory extensions. This can create memory hierarchies to be used by the Host and will likely require main changes to how the Host manage them, more complex than current NUMA memory management.

For the sake of what this paper tries to discuss, it is irrelevant the specific hardware/standard behind, CXL or equivalent, the focus being what this technology brings in for solving, improving and extending what SmartNICs do. Regarding the device type for an SmartNIC implementing a CXL-based solution as presented in this paper, a type 1 or type 2 could theoretically work, but there are other details like counters which could imply a type 2. It is not the goal of this work to go into that level of detail but just an intellectual exercise about the possibilities.

More specifically this paper covers what SmartNICs with Match and Action Tables (MATs) require and what a CXL-based solution could mean. The discussion is based on how Linux deals with the control path programmability, the support of offloading such control rules (match and action) and flow states (connection tracking) to SmartNICs and the necessity of having both, the slow path, through the operating system, and the fast path, through the hardware, as current OVS-TC offers. Other solutions as implemented by private offload mechanism could also obtain better results and simpler designs, but it is in the dual path where the benefits can be higher.

The Linux Traffic Control (TC) **[2]** was not designed for what SmartNICs try to solve in the massive virtualization area, with another Linux network component, netfilter, suffering similar scalability problems when hundreds of

thousands or even millions of connections need to be constantly tracked. This is not reported (or not loud enough) as a problem because the Linux kernel TC is the slow path, and it is not expected to be a performance issue since most of the traffic will be handled by the specialized SmartNIC. Of course, this is true for the case of such a hardware, but even in this scenario the current kernel implementation is arguably not good enough because the slow path can eat a considerable amount of cpu cycles, and any latency when offloading the control rules or flows will add up to the problem. The misconception is likely due to considering the setup for the control rules something static or quasi-static, meaning that happens once when the related VM is deployed, but this is not always the case. OVS default behavior is to add/remove the required control rules based on traffic, so the setup is dynamic and the rate of change dependent on the specific system needs and configuration. This is even worse for offloading flow states which can fully stress out the offloading mechanism if thousands or dozens of thousands of connections appear or disappear per second, as it can be expected with VMs acting as servers. If we consider the slow path as the only path, what is the case for supporting more rules/flows than what the hardware is able to cope with, or for those complex match and actions not fully supported by the HW, improving the slow path acquires more dramatic urgency. Although important by itself, this paper focuses on the potential benefits for the dual path when using a CXL-based design.

Interestingly, there is a related technology, P4 **[3]**, which can alleviate the scalability issues along with adding the promising land of a full control path programmability. This is not only in the hardware side P4 was originally designed for, but also in the slow path where the way packets are processed is (or should be) equivalent to what the hardware does, and due to hardware limitations, usually a superset of those control rules/flows. With a P4 frontend interface and a software P4 interpreter/compiler, the same functionality can theoretically be obtained, both, in the slow and fast paths. Leaving aside how this will be finally (if so) implemented, and assuming the performance will significantly improve (and therefore current TC and netfilter conntrack bottlenecks overcome) a following set of questions arise: could the offload of those match and action rules be also improved for minimizing the cpu required even with such an optimized new packet processing component? How can the current per driver vendor code be avoided for translating the equivalent TC/conntrack semantics to those per vendor proprietary? If same P4 code is being used, by software and hardware, could the same data tables in a single place/memory be used?

It could be argued that every vendor will implement the hardware MAT tables differently so this cannot be imposed or an agreement for a standard reached, and that it will lack the flexibility required. But it is not the specific and per-vendor hardware tables the ones to converge but the memory backing their contents. For an SmartNIC implementing those MATs hardware tables and focused on massive virtualization, just a small set of the rules or the flows to work with will be populated into those tables, with most of them being in device RAM memory. How the hardware tables can be populated based on the memory contents is up to each vendor, where several options are available, like reserving HW table entries for specific rules/flows/clients, implement specific eviction/scheduling where those HW table entries will be populated based on generic or specific necessities, or even to have two different fast paths inside the device, one being the fast-fast where the packets can be handled using the HW table entries, another where the packets are handled using the device memory. With rules and flows kept in Host DRAM for the slow path, and most of them also in the device DRAM, if we put a CXL-based design into the picture, same contents could be used by the device or by the main cpus in the slow path: just updating at specific memory addresses once and each side will use them. This is where the CXL device type will matter. A type 1 device would imply the Host memory being updated and the Device memory being a cache of those contents. In this case the slow path will get the entries to work with from Host memory. If a type 2 device is used, the CPUs will update the device memory, so cache misses when accessing rules/flows entries by the slow path would imply getting the contents from the device (transparently performed by the CXL protocol). As commented before, which type to use needs to be studied in more detail, and maybe that could be just a vendor decision.

While, as commented previously, device memory can be directly accessed from the Host cpus as any other memory if mapped accordingly (PCIe BARs regions or specific memory mapping in embedded designs), and surely some closed solutions make use of this possibility, with CXL there exist important improvements as already explained. In any case, the rules/flows entries in the device memory can be populated/depopulated by the Host CPUs with load/stores implying a smooth offloading process.

CXL can be a big change and make possible other functionalities arguably required for achieving the full dream of virtualization where full confidentiality can be provided, which is explored briefly in this document. If we put all this together, if the holistic approach is pursued, the standardization about how to write the memory is likely not needed at all: a fully programmable datapath would give cloud tenants the power of defining that themselves, where the rules to apply will be in sync with what the hardware supports. Even if the fully programmability is not achieved or not in the short term, what unsurprisingly seems to be harder in the hardware side, the slow path programmability could not need that standardization as long as the programmability is offered by each vendor. This is a more likely outcome in the short term than the fully programmable hardware datapath, and the CXL approach can be useful in any case.

In the next sections the problem with the current slow path implementation is described, then the specific problem of offloading rules/flows to SmartNICs when used in massive virtualization deployments. Next we will compare the current offloading process to a theoretical one based on CXL. It is worth to mention this design would

need the scriptable P4 (or something equivalent) in the slow path, what could be seen as a reinforcement of such an idea, or maybe a mutual reinforcement. Finally, a further exercise is performed looking at possibilities in the Openflow area, multitenancy support and per-client control path ownership, as another CXL potential disruption.

2. SmartNICs: MATs and DRAM

Terminology can always be confusing, so it is necessary to state what kind of SmartNIC we mean in this discussion. Servers can run dozens of virtual machines and code executed inside those virtual machines (VMs) can achieve native cpu execution except when I/O is required, which is emulated and therefore suffers from extra work needing cpu cycles.

SmartNICs can significantly improve this with the code inside the VMs doing I/O straight to the hardware instead of going through the host software emulation. The idea is to give each VM a chunk of the hardware, but then the problem is how the hardware knows what to do with the I/O coming from those VMs.

Reducing the discussion to networking, the hardware needs to have rules for handling those packets, and for figuring out how packets coming through the wire need to be dispatched to the right VM. For doing so efficiently, there are hardware CAM tables containing those rules to apply. A rule defines first which packets the rule should be applied to, and then which actions to execute for those packets, like redirecting them to a specific port or adding/removing a tunnel header.

Hardware CAM tables are how Match and Action Tables (MATs) can be more efficiently implemented, but they are power hungry beasts because the matching happens in parallel implying a lot of gates being involved at the same time. The dimension of those CAMs does not suit what fat servers require, not just in terms of number of rules but also regarding the flows to be tracked (if this service is offered what seems to be common nowadays). The solution is to back up the rules and flows with DRAM and use hashing for obtaining the rule or flow to work with. The relationship between those rules and flows in DRAM and the CAMs is up to the hardware design, but the DRAM will be used for a massive virtualization scenario. So, the question is how those CAM tables and that DRAM end up being populated with the rules and flows required.

The Openflow/OVS way

The Open Virtual Switch (OVS) **[4]** was designed following the Openflow **[5]** standard, the control and forwarding paths of an SDN **[6]** architecture. The idea being a centralized controller having the information about how a VM being deployed needs to be configured in terms of its networking communications. The host executing the specific VM will get that information from the controller through the OVS component which configures that VM networking accordingly.

OVS is implemented with two cooperative components, one inside the kernel and the other in user space. The one inside the kernel does the handling for those packets coming from the OVS ports connected to VMs and from the wire. If a packet did not match any configured rule, so no action can be applied, it is sent to the user space component. At that point the OVS user space component can know what to do with the packet or maybe it requires to ask the controller about it. Once it does know, a new rule specifying the handling will be configured into the kernel side and the packet injected back to the kernel for being handled. The in-kernel rule configuration implies to install it inside the data structures used by OVS which evolved through the different versions and it is well described in **[4]**.

Before talking about when hardware offload changed this picture, it is necessary to comment the rules to work with can be installed at the time of the VM deployment, then being a static rule configuration happening once. But this is not always the case and OVS by default keeps this dynamic based on the traffic itself, and rules being installed and removed constantly. That is why the way the rules can be updated, in software or in hardware, does matter, and any limitation in that regard having a significant impact on the full system, not only in the related VMs. This rate of change is also unarguably important when the conntrack functionality is considered along the rules.

While OVS helped to deal with the requirements for networking in virtualization, it is a software solution being executed in the same cpus used by cloud clients. Indeed the way VMs were attached to OVS was through I/O emulation requiring more cpu cycles to be stolen from the clients or just those cycles not being *monetizable*, and of course the performance suffers. Hardware came to the rescue where the OVS configuration could be applied and packets handled without host intervention. Or so once the rules are offloaded. The problem in the Linux world was how to perform such an offload. At the time there were private per vendor solutions but attempts to include them in the kernel were rejected. It was not the first time such kind of divergent implementations had to be added to the kernel with the usual solution being to make it a configurable option selecting which solution to use, but requiring, of course, some minimum convergence for allowing such flexibility. In this case it did not happen that way because it was considered the Linux Traffic Control (TC) component could be used with minor changes for supporting hardware offload. It required though the TC infrastructure being part of the OVS kernel component with the rules to match with and the actions to perform being those already present (or easily extended as it happened with TC flower).

3. TC and conntrack bottlenecks

As mentioned, offloading the control path was under discussion for several years and it was finally accepted as an extension to the Linux Traffic Control Infrastructure. It is easy to criticize that decision *a posteriori* but it can be arguably said that:

1. The number of rules/flows to support was considered but not its rate of change.

2. Any overhead makes things far worse than just a temporal degradation.

3. It was good enough for Diffserv [2] but not for massive virtualization.

As already commented, the rules to apply when a new VM is deployed are not (usually) static but dependent on network traffic. That is where the fast path/slow path appears, initially the fast path related to packet handling inside the kernel, the slow path meaning the packet needed to go to user space for deciding what to do. With the hardware offload the fast path is the hardware and the slow path the software handling by the host which included the kernel and user space handling. Therefore, how the rules to act on packets change, its rate of change, it does matter.
The second point is even more relevant for hardware offload because the latency to offload, the latency until the hardware can use the offload rule, implies further packets will need to go through the slow path, the host, which requires cpu cycles contending with the needs from VMs and with the slow path itself dealing with other flows.

Five problems with TC

The first problem with TC is it is not designed for updating a qdisc in parallel. Let's see how this can affect the rate of change with a vxlan device. VMs belonging to the same virtual network and running in different serves communicate through a tunnel which they are not aware of. This tunnel is deployed and handled by the virtual network provider and with vxlan tunnels a vxlan device is created and therefore a qdisc. It is in this vxlan qdisc ingress (after the UDP layer has detected the packet belonging to a tunnel connection) where the rules will be added, but because this sequential updating restriction, concurrent flows reaching the host and being handled by different cores through an efficient RSS (with inner headers for the case of tunneling) will contend for updating the qdisc. Not so long ago someone tried to submit a patch for adding parallelism, but it was discarded because it required locks which are not easy to handle inside TC, and because that is the control path after all, which does not require this kind of optimizations. With the right motivation this could change but this is not the only problem.

The second problem with TC is those qdisc rules, once they have been installed, will be checked out sequentially until one matches the packet. This is not so bad for a low number of rules to work with per qdisc, but all the cpu cycles required here are a problem with the current networking performance available with gigabit cards. Again, TC was designed for supporting the Diffserv architecture at a time where the needs for massive virtualization were distant and where the gap between the cpu packet handling and network cards performance was not so bad. User space solutions like netmap or DPDK appeared because this gap increased (along with the problems coming with system calls and interrupts). Should the kernel move to another model or maybe should it keep TC along with a more suitable solution for massive virtualization?.

The third problem is related to parsing and matching what is performed for each rule until one match is found. Other implementations dealing with matching packets are far better in this regard (see [4]) which comes from the necessity of dealing with a far higher number of rules to match with. Also, the current parser, kernel flow dissector, can have been useful until now or still up to the task for certain scenarios, but its performance, leaving its implementation clarity aside, highly improvable **[7]**.

A fourth problem with TC is the syntax gap. For massive virtualization and with OVS-TC relying on Openflow, those rules defined in Openflow syntax need to be translated to TC. This is even worse with hardware offload where such a TC rule will need to be translated to the specifics of each vendor by the related driver.

And the fifth problem with TC is its ossification. Interestingly this was something always favorably to software versus hardware solutions, but supporting quickly new protocols requires another approach. This could hopefully be overcome soon if the P4 software and hardware datapaths end up being a reality.

SmartNICs can alleviate these problems because the datapath will be in the hardware. Right? Well, the slow path is still there with SmartNICs, and here it comes the bad news: any latency offloading rules implies more packet through the slow path, which implies more latency for the offload again.

Conntrack design is probably better suited for supporting a higher load of processing, at least for checking if a particular flow is already being tracked. But conntrack as a TC action depends on the rules execution and therefore suffers from the sequential process.

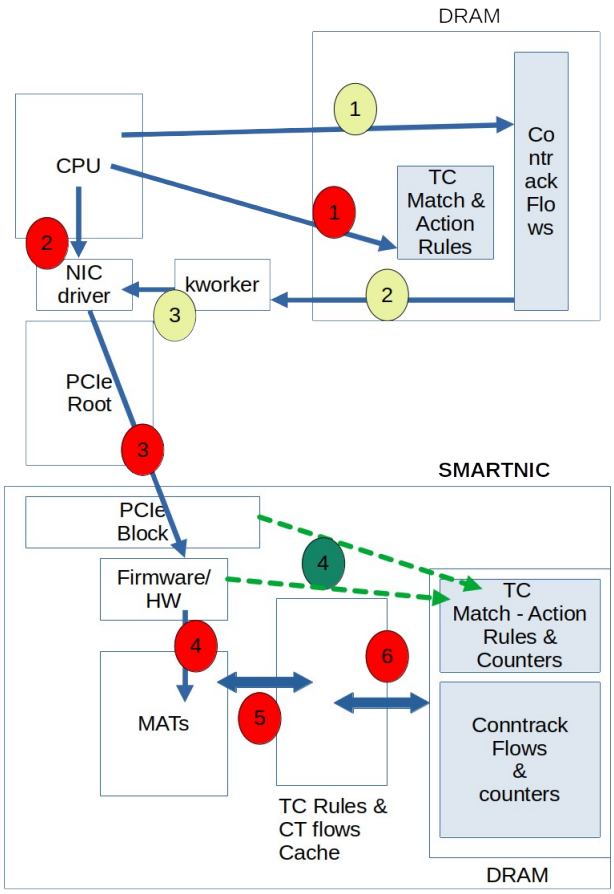
Offloading

TC rules and conntrack flow states can be offload to the hardware. The limitations exposed have an impact on the offload because it cannot happen faster than TC or conntrack add rules or flows. But the offload itself has problems.

A TC rule to offload needs to be received by the related hardware driver, what is currently happening at the end of the TC rule insertion, and this can not happen in parallel as we have mentioned in the TC problems list. Then the driver will check for the hardware supporting the rule requirements, and if supported, it will translate the TC rule to the format required by the hardware. Interestingly, data about the rule to offload is kept by the driver for dealing with new necessities (updating, removing, counters). At that point a specific per-vendor mechanism is used for sending the rule and, usually, waiting for the completion of the operation. This is of course up to the vendor where the possibilities are a specific hardware block, firmware, or both.

For conntrack the offload does not happen synchronously but through kworkers. Those works will, when processed by the kernel kworkers, end up reaching the driver and a similar process than with TC rules will occur, although usually simpler in this case. Currently the

implementation creates two different works for a flow offload, one for each direction. It is not necessary to say how the system will use the cpu for executing those kworkers will have an impact, and the fact of not existing the notion of processing flows batches makes this mechanism a problem with massive virtualization where conntrack flows will be inserted and deleted constantly based on traffic.



The previous picture tries to show the current processing when TC rules and flow states are offloaded. The Host needs to process not just the TC insertions and conntrack flow states changes but the specific driver will need to do more work before the data ends up in the hardware. The final latency has two components here, the latency introduced by the kernel and the latency introduced by the driver/hardware for exchanging the control data. There is another component and this is related to how the hardware will handle the control data until it is really used by the hardware datapath. There are different possibilities for implementing this final step, like redirection to DRAM or populating the MATs with the new data, and that can be done through specific hardware design or by firmware. But apart from the specific design, avoiding latencies here is not trivial since the mechanism needs to be designed for the rate of change required.

An SmartNIC with a DRAM will be constantly using the DRAM when fully operative, otherwise the device design/dimensioning is arguably inappropriate. If so, does it not make sense to use the direct DRAM population from the Host and with the best available technology? A counterargument is some rules/flows could be installed in MATs with lower latency through a direct-MAT population, but this requires a carefully and costly design if unexpected latencies needs to be fully avoided. If this minimal low latency is required, which can only be done up to a point (MATs size), some internal mechanism could be added for DRAM rules/flows entries triggering such a population based on certain flag or similar signal. Simplifying the control path inside the device has also the advantage of keeping the focus on the datapath, in this case focusing on how the interaction between MATs and DRAM, and any intermediate cache, can be improved.

4. CXL into the picture

It should be clear at this point which are the main problems with the current Linux way. Maybe it is also worth to discuss if the slow/fast paths is the best option. This paper assumes it is, so just a few lines to back this up:

- Overcoming the hardware resources limitation.
- Matching or actions not supported by the hardware.
- Flow initialization based on host-only functionalities like conntrack. Currently conntrack state can be offloaded but only when the state is established by the software.

As commented earlier, CXL does also make sense without the dual datapath since it can potentially (based on the hardware design though) minimize the latency of rules or flows offload, using less host cpu cycles and likely saving energy with a more efficient protocol than adhoc hardware designs.

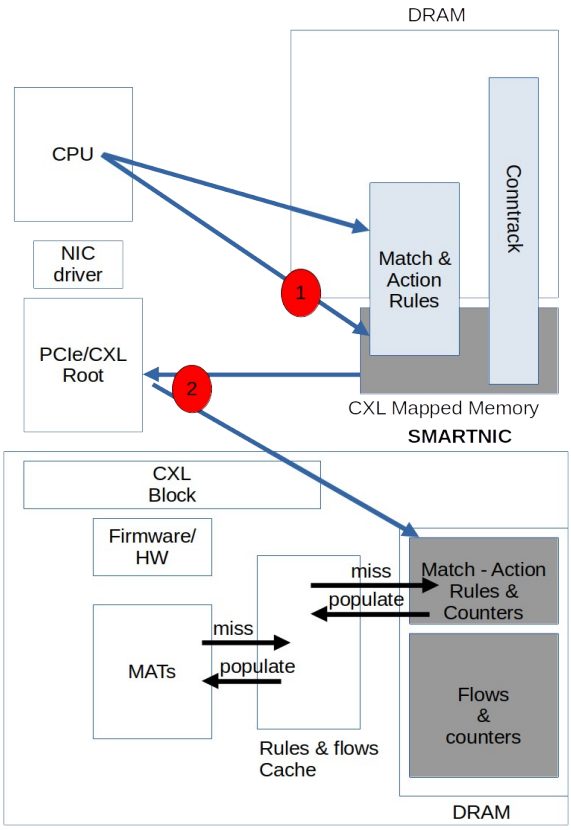
With the dual datapath, CXL could be used just for the offload, with vendor drivers writing to CXL memory addresses. This is the more conspicuous use and likely the first step towards a fully CXL support, but although it would benefit from the CXL protocol advantages, it would not exploit the possibilities. So, which are the extra gains for the dual path scenario?

CXL was designed for the requirements of sharing memory between devices and cpus where the coherency plays a key role. With the necessities of massive virtualization, the amount of memory required for keeping the rules and flows is significant. With the dual datapath, as it is currently implemented, that amount of memory doubles (or triples if the data kept by the driver counts), so could not the same memory contents be used by both, the hardware and software datapath?. This is theoretically possible but there are some caveats which require further discussion. Before that, a second potential advantage needs to be presented.

While a single path with an adhoc vendor solution can write to CXL memory the specific contents with no constraint, the dual datapath requires a generic way as a

frontend with specific per vendor drivers as backends. Removing this indirection is possible if there is just a generic way which implies agreement or standardization about what to write to CXL memory. It is important to note this is not about restricting or forcing vendors what to do since the secret sauce about the MATs design and any packet switching details can be kept: the only requirement is about how the rules and flows are saved in the DRAM. It could be argued that format will be intimately related to such internal design, but, in that case, who should be doing the required translation? If it is before the CXL memory is written, that will be charged on the host cpus. If it can be performed by each vendor inside the device when MATs or intermediate caches are populated, the burden will be on its vendor design. Moreover, could not be a fully P4 solution, where the hardware and software will depend on the P4 design, the one in charge of how to populate the DRAM? Finally, it is worth to question if the several translations performed today, the syntax gap, could not be avoided, at least for the hardware, if the devices would pursue Openflow compatibility. Could a fully P4 solution have the right format already in the Openflow controller?

Now, if we join the last two concepts, there it comes an interesting solution:



The picture is hopefully showing the design possibilities. The memory written by the Host, when rules or flows need to be updated, is a CXL memory. Once there is a change, thanks to the CXL coherency, both, the device and the Host, will see the same tables, not the MATs but the data structure to be used in DRAM. When the slow path is executed in the Host, the CXL coherency appears, with the cpu using CXL memory for data not present in the cpu cache. Those cache misses are likely handled with higher latency than accesses to the Host DRAM, although this depends on the CXL device type. If the data is in Host DRAM and the device is caching the data, the Host memory needs to be dimensioned properly. If the device DRAM keeps the data and the Host is caching, the dimensioning depends on the vendor. The data will only really be in one place in the second case, but options should be considered. As the picture also shows, there is part of the table not under CXL domain: overcoming the HW resources and supporting by software what the HW does not.

In a dual datapath scenario this design implies the hardware will get the new control data as soon as the beneath hardware allows and not dependent on middle layers adding latency and subject to potential bottlenecks. As we have shown previously, current offload mechanism through specific drivers are not needed, and hopefully not translations either. Is this an impossible dream?

The caveats do exist, but they are not unbearable. The most important one is with this design the software and hardware need to use same algorithm for accessing the data in the shared memory. The most likely solution seems to be a number of cuckoo hash tables as some current hardware switches are using. But this does not need to be fixed but through a configuration option based on what the HW requires. The initial option could likely be just an agreement following the current state of the art, but this design would surely lead to further research and other options appearing in the following years. As an example, another potential solution with CXL memory could be the control data not really written by the Host but by specific HW dealing with the intrinsics of the data structure used, where updates (as with cuckoo hashes) could be harder and therefore potentially accelerated/controlled by specific hardware elements. Of course, the software should have a read-only way for using the data in the slow path execution. The CXL advantage would still be there with the writes going to specific hardware buffers where those write/updates accelerators will use them. This is quite similar to how GPUs currently receive commands from the Host and the performance and design of those elements will set the latencies. Finally, and again, a full P4 solution involving the software and hardware datapaths could potentially include the way of accessing the data from both sides efficiently.

The host writing straight to the CXL memory has another important advantage: this is orthogonal to other interesting things the host needs to do (or could do). One of those things is yet another intellectual exercise about further CXL possibilities in the SmartNIC area, specifically now about how to support multitenancy with the proper confidentiality and performance and through a simpler interface than with adhoc solutions. This will be discussed in the next section but before that, let´s talk about hardware counters.

Reading counters on demand with CXL

Those working in the SmartNIC area through the last 10 years are well aware of how offering counters to the Host can be a nightmare by itself. Having multiple wire interfaces in a network card and giving stats about how packets are sent and received is a tractable problem but that explodes when you have dozens or hundreds of VFs plus hundreds or thousands of rules, actions and flows.

A common solution is to send the counters regularly to the Host which requires the related driver passing those counters to the related software counters. The data can be sent through a special channel or just using the datapath and some filter in the driver for detecting those special packets. The work is not negligible and the time between the counters being packed into those special packets until the interested user gets them can be significant. Moreover, all the work done could be for nothing if none is interested. So, what if those counters are read on demand?

Here is where the cache coherency could be interesting because the coherency is not needed until the user reads them and just those the user is interested in. The granularity of this coherency is important and something to look at, but the main problem to solve here is how the user knows about the memory address to read from, and more importantly, how the hardware should keep them. We are not going to discuss this further, but it is important to note the fact that we mentioned the user and not the kernel knowing about how to read the counters. The next section explains what.

5. Openflow and Multitenancy

The orthogonality of a CXL design could be the key for achieving a dream in the SDN world: the programmability of the control path under the client command.

So, what is still needed for achieving the network virtualization dream? The answer, although probably not complete, is threefold:

1) Two-level control path: cloud provider at the first level, tenants at the second one.

The Cloud provider has to give a network to the client, a virtualized one. But how the exchanges inside that network happen should not arguably be under the control of the cloud provider. Some clients will be happy to forget about it and have the work done somehow with a default control path configured by the cloud provider. But surely big clients do not want to leave such a control to the cloud provider because it gives a lot of information about what is being done by the client. Not just how the communications are being done but also what communications are not allowed is information none with security concerns is happy to unveil. Moreover, a security breach in the host could manipulate that control in nefarious ways.

This has been proposed in Openflow **[8][9][10][11]** for allowing clients to program routers slices assigned to them. We state that there exists the same need for those private slices with the SmartNICs control path being under the control of the client owning the VM. Note this does not mean under the control inside the VM.

2) Fully isolation and privacy about control path set/required by tenants inside their network.

Because the previous point, the control should not only be performed by the client but it needs to be protected. This includes the previous statement about a security breach in the host not able to modify the control path of a client, and also the fact that the control path inside the virtual network should not be visible for the cloud provider either.

This should not be confused with the privacy offered by encrypted tunneling protocols like IPSec which can encrypt not just the data but also the inner headers. The problem being the Host and the NIC need to know which are those inner headers for doing the proper forwarding after the tunnel encapsulation is removed. A proper solution is only possible if only in the client´s execution context can those inner headers not just be read but also used for the matching and action required, and of course, for configuring the matching rules. There is no doubt this is really demanding, but there should not be doubt either that only that way can be the protection and privacy really offered if this two-level network control management is really required.

3) Qos applied independently in the two levels.

While the cloud provider hopefully offers certain degree of QoS per VM/client, how the guaranteed bandwidth is internally used should be in the hands of the client. CXL can help facilitating access to related counters and maybe using a per rule/flow weight which can be easily modified by the client and by the provider. This does not mean the provider can modify what the client does, but obviously, the provider needs to have also a way for doing the QoS at its level.

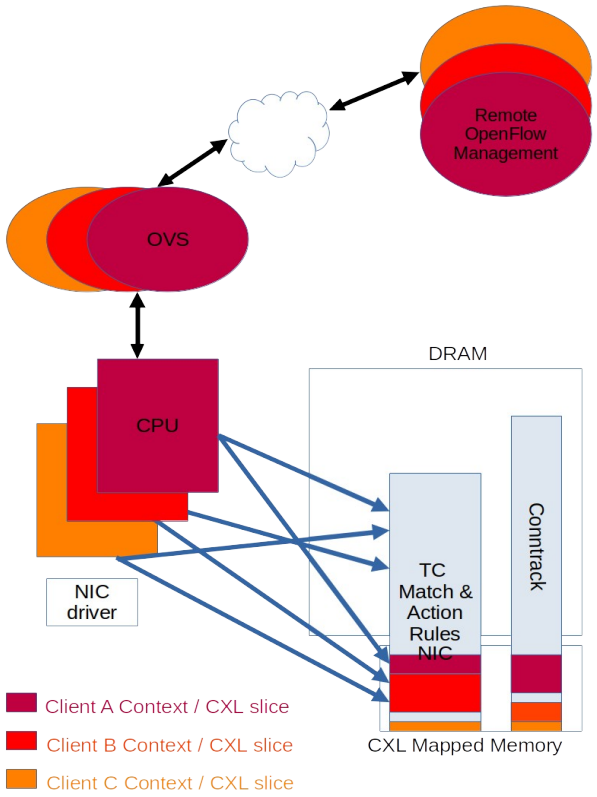
How CXL can help here? The Holistic Approach.

Because the configuration can be based on Host CPU writes to CXL memory, the different levels can have private mappings to different slices in that memory. There is no need of code in the Host translating generic configuration about control path to the specific instrinsics of a particular hardware device. This could be also possible without CXL, but the interface which is required for each level (the client level implying multiple clients) is simpler with just CPU read/writes to the private CXL memory slice, and as it was presented earlier, not requiring a special path through kernel drivers, nor special PCIe management in the device for the control path changes ending up in the device DRAM.

It is an holistic approach because with a memory interface plus real memory in the device, memory encryption technology can be used for the protection and privacy required. Technologies like AMD´s SEV-SNP **[12]**, Intel´s SGX **[13]** or ARM´s CCA **[14]** could be leveraged for such a confidential control path data.

And as with CXL counters are theoretically easily accessible helping with adjusting QoS configuration, and

with such a configuration having the simple interface of memory accesses in the two levels.



The previous picture shows how this could be implemented. The different colors for the CPU´s tries to reflect the different execution contexts where the CXL memory, specifically those ranges linked to the context, will be accessed, both for the slow path execution and for the updates to the rules and flows (the control path).

One context executing OVS and installing rules and flows under the control of the cloud provider is needed since the virtual network is created by the cloud provider and how to connect with other local or remote VMs is established at that level (how the defined tunnels are given to the client after this setup is not discussed here but not a big issue if this solution ends up being a reality). There is also one remote OpenFlow management system owned and managed by the cloud provider. There are other contexts, one per tenant having a VM in the machine, with specific CXL slices owned by tenants. Writing or reading the CXL slices is only possible inside the execution context owning such slice. The Openflow management is visualized with different systems with each tenant controlling its own one.
With those encryption technologies commented previously, those CLX slices and the slices in the main host memory used (for supporting more rules/flows than the hardware admits) will be encrypted protecting the contents from a security breach or from an non-trusted provider. Executing code writing to or reading from CXL memory slices is feasible, although current kernel-based solutions will require important changes. As an example, there are epbf programs nowadays handling packets inside the kernel implying special contexts (ebpf virtual machines) where that code is executed, so this specific CXL context could also be theoretically supported. When should those context be executed? This is obviously something requiring further investigation and, undoubtedly, discussions with the Linux kernel community and related projects like OVS. But, currently, the slow path is not counted on the client but on the Host. Should not be fairer to have those contexts contending with the cpu cycles assigned to the related client´s VMs?

There will be those arguing the cost of doing this context-based execution for handling packets will slow down the performance significantly, but this is only true if compared with a fast path not requiring such slow path functionality. Most of the fast path will happen inside the NIC with the offloaded rules and flows already installed through the CXL interface, and although as stated previously in this document, the slow path should be optimized as much as possible, the trade-off between the security and privacy and the potential performance degradation seems to be good enough **[15]** for the security side. Moreover, if this solution is considered as the right one, and there is a demand for it, the performance would be improved with specific hardware designs by the different CPUs architectures keen to offer this possibility, as it has happened in the past.

It is also important to note the privacy offered with the solution described in this document should cover all the involved components. The Host, being it the main OS, the hypervisor or other VMs/clients, should not be able to alter the control path of a specific VM/client nor to see its configuration, but the NIC should not break this privacy either. This is obviously a complex thing to do but not different to what is being done for preserving the VMs privacy nowadays and therefore not just feasible but desirable for the sake of giving the full privacy pack required in public clouds. This could be implemented partially or fully. Partially implying encryption is not used inside the NIC in all the processing, although it is obvious the right keys will be needed at some point, but the client´s related data should not be readable from NIC memories or registers without the client collaboration in special cases like debugging.

6. Conclusions

There is no doubt CXL is a paradigm change and it will bring new possibilities only limited by the inventive to harness it in unexpected ways.

With an CXL Type 1 or 2 SmartNIC, the Linux way could have these advantages:

1. Data in the slow path with rule and connection entries shared with the hardware fast path.

2. Population and depopulation by the Host with just memory operations by the CPUs. Offloads do not require extra work.

3. Counters can be read, and coherency can be delayed until required.

4. Arguably, CXL-based setup can facilitate Device resource management and QoS from the Host.

5. Future HW designs based on fully programmable pipelines, as scriptable P4 points to, or an equivalent HW eBPF, would suffer same problem than current solutions regarding population and depopulation of MATs and hash tables. The rate of change for the control data will be higher than conventionally assumed, and a CXL-based design can only help.

6. An holistic approach with CXL along with memory encryption technologies can make possible the full virtualization dream, or at least being one (giant) step closer.

This proposal is really ambitious and no doubt it requires the SmartNIC vendors to believe in the prospects, although maybe it is the cloud providers, knowing better how could be the impact of such technology, the ones pushing forward. Being aware of the challenges ahead, we use the same initial paragraph, this time applied to the proposal itself:

New technologies can always be disruptive but only if achieving the expectations once the marketing dust settles down and the advantages can clearly be seen without the blurry vision of just-to-prove promises.

7. References

[1] CXL https://www.computeexpresslink.org/download-the-specification

[2] *Linux Traffic Control - Implementation Overview* *https://api.semanticscholar.org/CorpusID:60992052*

[3] Your Network Datapath Will Be P4 Scripted *https://www.netdevconf.org/0x16/slides/38/P4TC-0x16.pdf*

[4] The design and Implementation of Open vSwitch *https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff*

[5] OpenFlow., January2014. *http://www.opennetworking.org/sdn-resources/onf-specifications/openflow*

[6] B. Naudts, M. Kind, F. -J. Westphal, S. Verbrugge, D. Colle and M. Pickavet, "Techno-economic Analysis of Software Defined Networking as Architecture for the Virtualization of a Mobile Network," 2012 European Workshop on Software Defined Networking, Darmstadt, Germany, 2012, pp. 67-72, doi: 10.1109/EWSDN.2012.27.

[7]https://netdevconf.info/0x15/slides/16/Flow%20dissector_PANDA%20parser.pdf

[8] Higuchi and T. Hirotsu, "Design and Implementation of Virtual Topology Management for Multi-tenant OpenFlow Hypervisor," *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, NV, USA, 2017, pp. 1523-1528, doi: 10.1109/CSCI.2017.266.
https://ieeexplore.ieee.org/document/8561030

[9] M. Erel-Özçevik and B. Canberk, "OFaaS: OpenFlow Switch as a Service for Multi Tenant Slicing in SD-CDN," in *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 362-373, March 2021, doi: 10.1109/TNSM.2020.3045044.
https://ieeexplore.ieee.org/document/9295349

[10] C. Argyropoulos, S. Mastorakis, K. Giotis, G. Androulidakis, D. Kalogeras and V. Maglaris, "Control-plane slicing methods in multi-tenant software defined networks," *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Ottawa, ON, Canada, 2015, pp. 612-618, doi: 10.1109/INM.2015.7140345.
https://ieeexplore.ieee.org/document/7140345

[11] N. Paladi and C. Gehrmann, "Towards Secure Multi-tenant Virtualized Networks," *2015 IEEE Trustcom/BigDataSE/ISPA*, Helsinki, Finland, 2015, pp. 1180-1185, doi: 10.1109/Trustcom.2015.502.
https://ieeexplore.ieee.org/document/7140345

[12]https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[13]https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html

[14]https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture

[15] Performance Analysis of Scientific Computing Workloads on Trusted Execution Environments. Ayaz Akram, Anna Giannakou, Venkatesh Akella, Jason Lowe-Power, Sean Peisert. 2020ArXiv201013216A.
https://arxiv.org/abs/2010.13216