An Evaluation of Using CCIX for Cache-Coherent Host-FPGA Interfacing

Sajjad Tamimi, Florian Stock, Andreas Koch
Embedded Systems and Applications Group
TU Darmstadt
[surname]@esa.tu-darmstadt.de

Arthur Bernhardt, Ilia Petrov
Data Management Lab
Reutlingen University
[firstname].[surname]@reutlingen-university.de

***Abstract*—For a long time, most discrete accelerators have been attached to host systems using various generations of the PCI Express interface. However, with its lack of support for coherency between accelerator and host caches, fine-grained interactions require frequent cache-flushes, or even the use of inefficient uncached memory regions. The *Cache Coherent Interconnect for Accelerators (CCIX)* was the first multi-vendor standard for enabling cache-coherent host-accelerator attachments, and already is indicative of the capabilities of upcoming standards such as *Compute Express Link (CXL)*. In our work, we compare-and-contrast the use of CCIX with PCIe when interfacing an ARM-based host with two generations of CCIX-enabled FPGAs. We provide both low-level throughput and latency measurements for accesses and address translation, as well as examine an application-level use-case of using CCIX for fine-grained synchronization in an FPGA-accelerated database system. We can show that especially smaller reads from the FPGA to the host can benefit from CCIX by having roughly 33% shorter latency than PCIe. Small writes to the host have a latency roughly 32% higher than PCIe, though, since they carry a higher coherency overhead. For the database use-case, the use of CCIX allowed to maintain a constant synchronization latency even with heavy host-FPGA parallelism.**

I. Introduction

When combining conventional software-based processing on host CPUs with specialized hardware accelerators to perform heterogeneous computing for higher performance or better efficiency, the nature of the interface between the host and the accelerators is a key design decision.

For most discrete accelerators, such as GPUs or FPGA boards, PCI Express (short: PCIe) has long been the dominant interface. Its performance has steadily increased, with PCIe 4.0, the latest widely deployed version, reaching 1.97 GB/s per lane. However, PCIe is mostly optimized for high-throughput *bulk* transfers. E.g., as shown in [1], transfers of 128…256 KB are required to reach at least 50% of the theoretical bandwidth. The achievable throughput drops dramatically for the smaller transfer sizes (down to cache-line size) required for fine-grained host-accelerator interactions. And while PCIe has added extensions such as Address Translation Service (ATS) / Page Request Interface (PRI) to support shared virtual memory, or atomic operations, most implementations do not encompass cache coherency mechanisms.

978-1-6654-8332-2/22/$31.00 ©2022 IEEE

This makes fine grained interactions quite costly, as either cache *flushes* are required on both the host or accelerator sides when synchronizing execution or exchanging small parameters or results, or memory regions used for data transfers have to be marked as *uncached*, slowing down the accesses of the processing element (host or accelerator) they are physically located with.

To address this problem, a number of interfaces and protocols have been proposed that also cover cache coherency. In this work, we examine the use of Cache Coherent Interconnect for Accelerators (CCIX), the first of these interfaces specified as a multi-vendor standard and implemented across multiple different accelerator and host architectures. Further improvements can be expected in the near future once protocols such as Compute Express Link (CXL), which has even wider industry support, make their way into the market.

We present both detailed low-level measurements for various CCIX access scenarios, as well as an application-level use-case. The latter employs CCIX for high-performance synchronization between an FPGA accelerator and the host when running an Database Management System (DBMS) exploiting Near Data Processing (NDP). To our knowledge, this is the first time a cache-coherent accelerator interface has been used for that purpose.

We give an overview over some of the interfaces and protocols in the next section and then discuss CCIX details, especially in context of FPGA accelerators, in Section III. Our main contribution, though, is the evaluation, where we present both low-level characteristics in Section IV, as well as the application-level use-case in Section V. We conclude and look forward to future work in Section VI.

II. Related Work

a) PCIe: PCI Express [2] is the standard for connecting peripherals to desktop and server systems. PCIe scales by bundling multiple lanes for a single device to increase the bandwidth of the link. In version 1.0, it was capable of transferring 250 MB/s per lane. With each successor version the bandwidth roughly doubled, reaching now 7.88 GB/s per lane in version 6.0. Currently, version 6.0 is just specified, while hardware for 5.0 is upcoming, and 4.0 is the version most widely deployed on current hardware. PCIe uses full-duplex serial links, with a point-to-point topology that has two

additional layers on top of the electrical link layer, namely the data link and transaction layers. These additional layers provide error correction and packet-based communication. In addition to basic operations, like transferring data, initialization of devices etc., PCIe also supports more advanced (optional) features, such as PRI and ATS, but does not cover cache coherency.

b) CCIX: CCIX [3], [4] is an advanced I/O interconnect that enables two or more devices to share data in a *cache-coherent* manner. On the physical layer, it can be compatible with PCIe (though it optionally allows higher signalling rates), and only differs in the protocol and the endpoint controllers. It was introduced in 2016 by the CCIX Consortium, which was founded by AMD, ARM, Huawei, IBM, Mellanox, Qualcomm, Xilinx [5]. CCIX has been implemented on both ARM-based as well as x86-based CPUs.

c) Other Shared-Virtual-Memory (SVM) or Cache-Coherent SVM Interconnects: CCIX is not the only contender for shared virtual memory interconnects. Alibaba Group, Cisco Systems, Dell/EMC, Facebook, Google, HPE, Huawei, Intel, and Microsoft proposed the CXL [6] in 2019, based on prior work by Intel. While CCIX can run on older PCIe connections, CXL was designed initially based on PCIe 5.0. Thus, CXL can reach up to 32 GT/s (that is 3.94 GB/s) per lane, it provides similar functionality as CCIX, but uses a different logical view. CXL has seen even wider industrial uptake than CCIX and is expected to become the predominant solution in the years to come.

Another option is the Coherent Accelerator Processor Interface (CAPI, later OpenCAPI) introduced in 2014 by IBM. While the first version was also implemented on top of PCIe, recent versions a vendor-specific interface. CAPI is primarily used in IBM POWER-based hosts and thus has a more limited scope than CCIX and CXL. In OpenCAPI 3.0 (x8 lanes), it provides a bandwidth of 22 GB/s with read/write latencies of 298/80 ns [7].

While not a straightforward extension to PCIe like CCIX, another interconnect that supports cache coherent protocols is Gen-Z [8]. It provides up to 56 GT/s per lane, and allows, similar to PCIe, the combination of multiple lanes. Despite its promising features, no Gen-Z hardware has been commercially released and the technology will be merged into CXL.

d) Database acceleration on FPGAs: [9] gives a good overview of using FPGAs to accelerate database operations. The most common approach, e.g., as used in state-of-the-art solutions such as *Centaur* [10], employs FPGAs as offload accelerators for large-scale filtering, sorting, joining, or arithmetic computations. This mode of operation carries the cost of large data transfers from/to the FPGA, though, and is different from the *near-data processing* approach investigated here, which aims to avoid these transfers.

III. CCIX Architecture and Use on FPGAs

This section will give an overview of the general CCIX architecture and discusses how it can be used in two different FPGA families.

A. General Overview

Devices attach to CCIX at endpoints. For the discussion here, the relevant kinds of endpoints are the Home Agent (HA) and the Request Agent (RA). The HA acts as the "owner" of physical memory, to which it provides coherent access, while the RA performs non-local reads and writes to remote memory by communicating with the owning HA. What distinguishes CCIX from PCIe is that the RA can provide its own caching, but maintains *coherency* with the HA over CCIX. On the HA side, a change in the cache state will be propagated to the accessing RA(s) by sending appropriate messages. CCIX natively uses *physical* addresses for the accesses, but can optionally employ existing PCIe mechanisms to allow accelerators to employ *virtual* addresses. To perform the actual address translation, CCIX relies on the PCIe ATS mechanism, which is one of the reasons that CCIX-attached accelerators also maintain a conventional PCIe connection to the host on a different PCIe Virtual Channel (VC). Of the various CCIX topologies, which include meshes and switched hierarchies, we employ a simple one relying on direct connections between host and accelerator. Also, since all of the required operations including address translation and coherency are supported at the hardware interface level, no special device drivers or custom firmware is required on the host.

Fig. 1-(A) shows the high-level architecture of the cache-coherent host-FPGA attachment for CCIX-enabled devices. This block diagram has the host at the top and accelerator on the bottom, both connected via a CCIX-capable PCIe interface. CCIX uses multiple VCs on the PCIe transaction layer to carry both PCIe and CCIX traffic over the same PCIe slot. On the CCIX-capable slots, the transaction layer uses VC0 for PCIe-packets and VC1 for CCIX-packets, sharing the same physical and link layers. However, CCIX can optionally use Extended Speed Modes (ESM), which increase the signalling rate. For the PCIe 4.0 attachment we use, ESM increases the rate from 16 GT/s to 25 GT/s, with 128 payload bits per transfer. The ESM mode will be enabled automatically during the CCIX discovery phase at boot time if both parties (i.e., RA and HA) support it.

B. FPGA RA using Xilinx XDMA

Xilinx Virtex UltraScale+ HBM devices are CCIX capable, but have to implement the CCIX functionality as reconfigurable "soft" logic in the form of an extended XDMA IP block. As shown in Fig. 1-(B), the key blocks include a CCIX-capable PCIe-controller, an ATS switch, and a PCIe-AXIMM-bridge. The ATS switch is used to insert virtual-to-physical address translation requests into the regular PCIe communication via PCIe VC0, and later retrieve their results. It also includes a small Address Translation Cache (ATC) to buffer existing translation results, in order to avoid the relatively costly address translations for already known mappings. The AXIMM bridge provides memory-mapped communication between the host and the accelerator (mainly control-plane traffic). For data-plane accesses, the accelerator employs an on-chip cache realized using the Xilinx System Cache IP Block [11], which

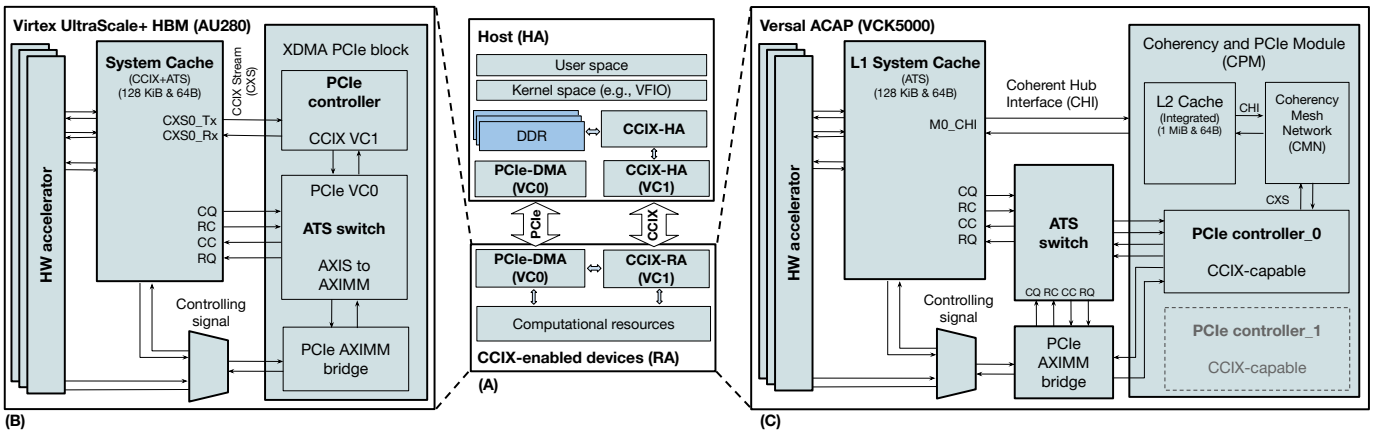

Fig. 1. Middle (**A**): Architecture of CCIX-capable host, acting as HA, with a CCIX-attached accelerator acting as RA. Left (**B**): SoC realizing CCIX-RA on a Xilinx *UltraScale+ HBM* device. Right (**C**): SoC realizing CCIX-RA on a *Versal ACAP* device.

in turns interacts with the CCIX coherency mechanisms using the CCIX streaming protocol. Misses in this cache become remote memory accesses, which are forwarded via CCIX to the HA to retrieve the data. The HA, in turn, ensures the coherency of the FPGA-side SC with the host-side caches.

C. FPGA RA using Xilinx CPM

The more recent Xilinx Versal devices have optimized "hardened" support for CCIX in their silicon. Specifically, the Coherency and PCIe Module (CPM) IP block [12] includes an integrated L2 cache, communicating with the chip-wide coherency mesh network using ARM's CHI protocol, which in turn again uses CXS to interface with the CCIX-capable PCIe controller. As before in the UltraScale+ device, two PCIe VCs are used to separate PCIe and CCIX traffic running over the same PCIe slot. Our setup requires only one of the two CCIX-capable PCIe controllers provided by the CPM block. The ATS Switch and AXIMM blocks are used as before.

D. Address Translation

After the System Cache (SC) receives a read/write request from the accelerator(s), it checks the ATC for the virtual-to-physical mapping. If the SC does not find a valid translation in the ATC (i.e., ATC miss), it requests the translation from the host using the PCIe ATS feature via VC0. The ATS interface on the system cache uses a request-completion protocol [13] to provide the translation service via four streaming interfaces: Incoming Completer Request (CQ), outgoing Completer Completion (CC), outgoing Requester Request (RQ), and incoming Requester Completion (RC). The reply from the host, e.g., holding the physical address, is delivered back to the FPGA using the same mechanism.

E. CCIX Timing Model

The average latency for a CCIX transaction is shown in Eq. 1. The latency for each transaction depends on the probability of having a valid cached address translation available in the ATC vs. having to request a new translation from the host by ATS, and then whether the requested data is present in the local on-chip cache vs. having to be requested from the remote HA. Note that with the use of ESM, the physical CCIX latency can be shorter than the physical PCIe latency.

$$\begin{aligned}\text{Latency} = {} & \text{P}_{\text{ATC-hit}} \times \text{Latency}_{\text{ATC}} \\ & + (1 - \text{P}_{\text{ATC-hit}}) \times \text{Latency}_{\text{ATS}} \\ & + \text{P}_{\text{cache-hit}} \times \text{Latency}_{\text{r/w-local}} \\ & + (1 - \text{P}_{\text{cache-hit}}) \times \text{Latency}_{\text{r/w-remote}} \\ \text{Latency}_{\text{r/w-remote}} = {} & \text{Latency}_{\text{CCIX}} + \text{Latency}_{\text{HA}}\end{aligned} \tag{1}$$

TABLE I
SPECIFICATIONS OF CCIX-ENABLE DEVICES

CCIX-capable devices	Core specification	PCIe specification
N1-SDP (HA)	4xARM N1-CPU, 16GB RAM, L1I (64KiB) per core, L1D (64KiB) per core, L2D (1MiB) per core, L3D (2MiB) shared	1x CCIX Gen4 x16 1x PCIe x16 Gen3
AU280 (RA)	L1 (128KiB, 64B)	1x CCIX Gen3 x16
VCK5000 (RA)	L1 (128KiB, 64B) L2 (1MB, 64B)	1x CCIX Gen4 x8

IV. Experimental Setup and Evaluation

We perform the practical evaluation in real hardware, namely using a CCIX-capable ARM N1-SDP platform as host, and Xilinx Alveo U280 (AU280) and VCK5000 CCIX-attached boards, having an UltraScale+ HBM and Versal ACAP FPGA, respectively, as the accelerators. Table I shows the specifications of the different devices.

A. Measurement Setup

All of the low-level benchmarks described later use the same basic measurement approach, which consists of three main components: a software Application Programming Interface

(API), a hardware module and the on-chip CCIX components described above. The software API runs on the host and is responsible for executing the benchmarks and reading the hardware-profiled CCIX latency characteristics. The software API has four main tasks: a) allocating a buffer in the host memory, b) initializing the hardware module with the access to measure, c) retrieving the latency data recorded by the hardware module, and d) profiling results. The pseudo-code of software API is shown in Algorithm 1. Note that we randomize addresses to force SC misses, thus ensuring that the CCIX transfers we are interested in actually take place.

Algorithm 1: Pseudo-code for software API in the ARM machine

```
Function host_API
    buffAddr = allocateBuffer();
    newAddr = buffAddr + rand();
    start_CTG (newAddr, length);
end
Function start_CTG (addr, length)
    configure_CTG(addr, length);
    wait();
    read_Register(&latency);
end
```

The hardware module, known as CCIX Traffic Generator (CTG), uses the fetch/store approach to capture CCIX latency. This module accepts requests (including type, virtual address, and length) from the startTrans call of the software API in the host. Following the API request, the CTG creates requests via the AXI4-MM interface to the SC, which performs the role of the CCIX RA, and then times the arrival of responses at the SC. The captured timings can then be read by the software API. Note that we consider a transaction complete only after *all* of its data has arrived.

Table. II shows the FPGA resources required for the simple CCIX-RA we examined. As shown in Fig. 1-(C), the VCK5000 uses a PCIe controller in the form of the hardened CPM module, but still needs some additional "soft" logic to support PCIe transfers and ATS translations.

B. Low-Level Experimental Evaluation

Experiment 1: CCIX vs PCIe - Latency and Throughput. In this experiment, we compared CCIX and PCIe transfer latencies for the relatively small block sizes of 32B to 16KiB typical for fine-grained interactions (and much smaller than the PCIe bulk transfers examined in [1]). The open-source TaPaSCo [14] framework was used to test the DMA transfers. In this experiment, the ATS latency was eliminated by making sure that the translation of the address is already existing in the ATC. Fig. 2-(A) and Fig. 2-(B) shows read and write latencies for PCIe and CCIX traffic, respectively. For the PCIe-DMA transfers, we used the high-performance DMA engine of TaPaSCo by setting different data transfer sizes and directly used the physical address of the data of the host memory. For the CCIX measurement, a buffer is allocated in host memory and its virtual address is passed to the CTG module.

TABLE II
CCIX-RA RESOURCE USAGE FOR AU280 AND VCK5000 AT 250 MHZ

CCIX-capable devices	Core specification	LUTs	Registers	BRAM
Alveo U280	Available resources	1303680	2607360	2016
	PCIe Block	3.53%	2.49%	3.17%
	System Cache	12.10%	3.54%	6.25%
	CTG	0.14%	0.07%	-
	ATS monitor	0.49%	0.16%	-
VCK5000	Available resources	899840	1799680	967
	CPM	0.02%	0.09%	4.14%
	PCIe Block	1.96%	0.76%	3.41%
	System Cache	9.24%	4.14%	8.38%
	ATS Switch	0.64%	0.44%	-
	CTG	0.24%	0.09%	-

TABLE III
COMPARING CCIX AND PCIE READ/WRITE THROUGHPUT ON THE AU280

Size (KiB)	Read-CCIX (GiB/s)	Write-CCIX (GiB/s)	Read-PCIe (GiB/s)	Write-PCIe (GiB/s)
1KiB	0.400	2.592	0.121	0.394
2KiB	0.487	4.065	0.152	0.883
4KiB	0.482	4.212	0.256	1.455
8KiB	0.486	3.145	0.354	2.942
16KiB	0.514	3.079	0.396	3.780

Our evaluation shows that on both the AU280 as well as the VCK5000, the CCIX transfers have a better latency for *reads* from the host in comparison to the PCIe-DMA transfers, as long as the data transferred is shorter than 4 KiB. In both cases, the speedup is due to the optimized packet protocol that is used by CCIX. However, when *writing* to the host memory from the FPGA using the optimized packet protocol, CCIX incurs longer latency compared to the PCIe transfers, as these writes participate in the coherency mechanism. Our throughput measurements show a read throughput for CCIX, relative to PCIe, of 3.3x, 1.29x, 0.87x for data set sizes of 1KiB, 16KiB, and 32KiB. Additional data points for read and write throughput are shown in Table III.

Experiment 2: Cost of ATS. The capability to transparently resolve virtual addresses simplifies the accelerator design and host interface considerably. However, that operation can be costly, as it might trigger a slow full page table walk on the host if the translation requested is not present in one of the host IOMMU's TLBs. In **Experiment 1**, we examined accesses that did not need address translations (noATS). But to examine the cost of ATS, we have now constructed two access scenarios, shown in Fig. 3: In the first (withATS), we force misses in *both* the SC and ATC, thus always incurring ATS overhead. In the second (noATS), we allow ATC hits, but still force an SC miss in order to actually have a CCIX transaction take place. The results show that especially for smaller transfers, the ATS overhead can be significant, leading to a *tripling* of access latencies on an ATC miss. For transfers of 32KB and more, though, the transfer time begins to dominate the ATS overhead.

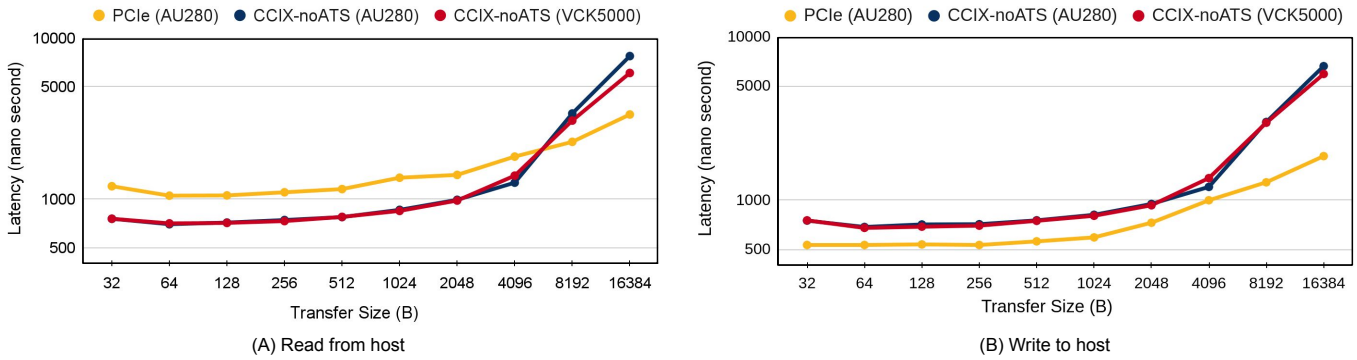

Fig. 2. Comparing CCIX and PCIe read/write access latency on the AU280 and VCK5000

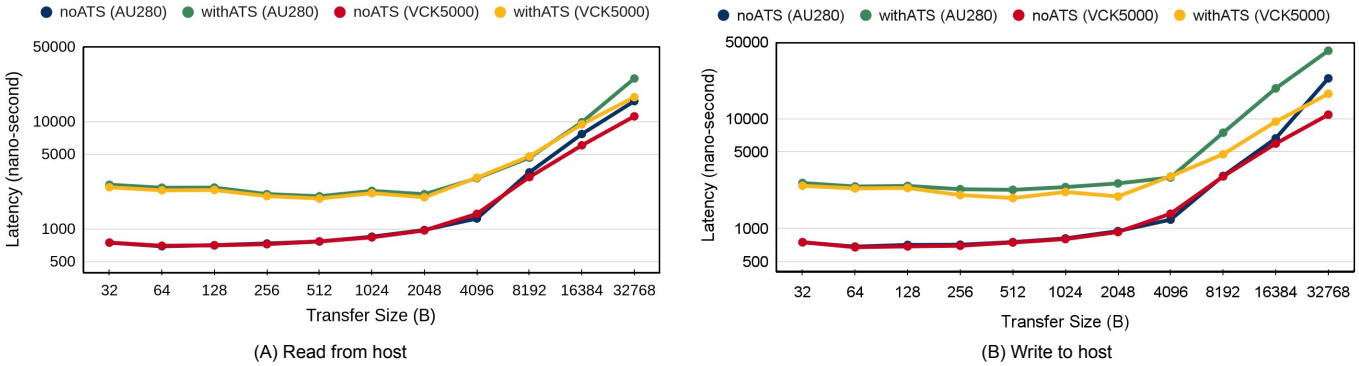

Fig. 3. ATS-effect on the CCIX access latency for random accesses to the RA module from CTG module on Alveo U280 card and VCK5000

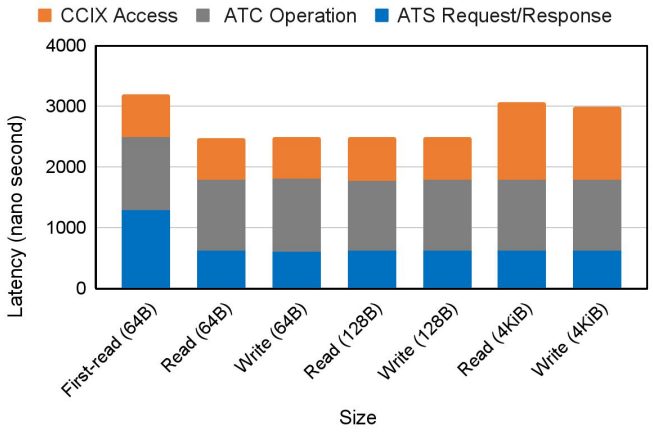

Fig. 4. Comparing read/write latency of CCIX-RA and ATS latency on Alveo U280 card

To further investigate the ATS latency, we can exploit the fact that the entire ATS mechanism is implemented in the ATS Switch block of the SoC. We can thus monitor the request/reply interface of that module to capture exact request-response timings for the ATS operation itself. Fig. 4 shows the CCIX access latency for blocks of 64 B (cache line size), 128 B, and 4 KiB. Since the Linux PageSize is 4KiB, these requests required only a single ATS translation each. By increasing the size of the request, more translations are needed. The initial access to the allocated buffer in host memory has the longest latency. Later sequential accesses have less ATS overhead, even when crossing onto another page at 4 KiB. We assume that this is due to pre-translations being performed by the host-IOMMU for the sequential accesses used here. For the case of a repeated 64 B read, by comparing the latency that is required by the host IOMMU to answer an ATS request ($\approx$ 617 ns, captured at the ATS Switch), and the known latency for a 64B read under an SC miss ($\approx$ 700 ns, from Fig. 3-(A)), the ATC itself appears to take $(2453 - 617 - 700 \approx 1136$ ns) for its operation.

One way to improve the latency of CCIX traffic is to mitigate the effect of address translation. This can be achieved, for example, by using the Linux huge page support. This will lead to larger pages, that, in turn, will have fewer page boundary crossings requiring fresh translations. The N1-SDP platform does support huge pages with different sizes (i.e., 64KB, 2MB, 32MB, and 1GB) at the boot time. We employed this approach in the database use-case (Section V) to improve performance.

Experiment 3: Data-Locality. The use of CCIX allows the accelerator to use its own cache(s), in confidence that they will always be coherent with the host. To show the best-case baseline performance of the two SoCs we evaluated the case where all accesses are guaranteed to *hit* in the on-device

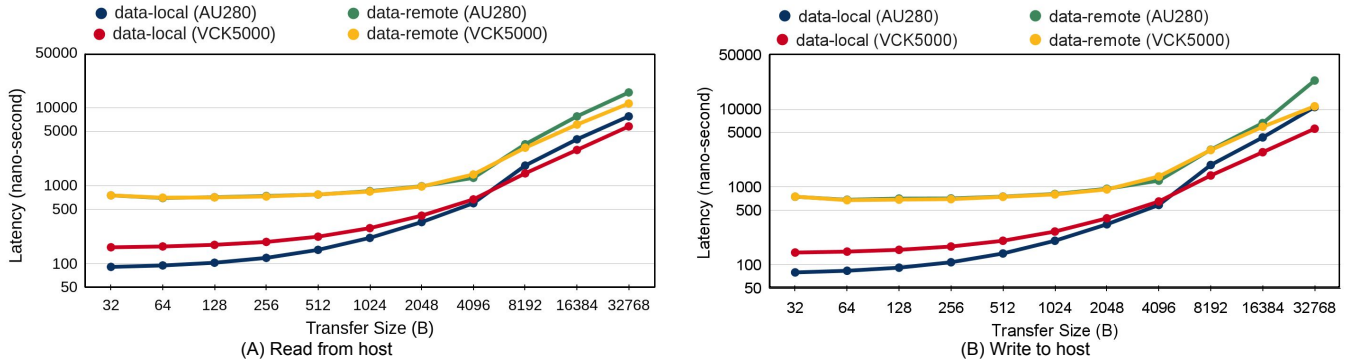

Fig. 5. The effect of data locality on the CCIX latency for AU280 and VCK5000

cache(s), called data-local in Fig. 5, and measure the latencies of these hits. For comparison, we also show the data-remote case which covers cache misses. The simpler cache hierarchy in the AU280 achieves smaller latencies (write $\approx$ 80 ns, read $\approx$ 100 ns) than the two-level one on the VCK5000 (write $\approx$ 150 ns, read $\approx$ 170 ns) for smaller transfer sizes. For larger transfers, though, the two-level hierarchy becomes faster.

Experiment 4: Coherency Efforts. In this scenario, the application on the host allocates a shared buffer that is *simultaneously* accessed and modified both by the host and the accelerator. These concurrent accesses/modifications increase the coherency efforts, and in turn, the access latency. A huge page is used to avoid ATS overhead. As outlined in Algorithm 2, the hardware CTG and the software API simultaneously modify cache lines in the shared buffer. Initially, we use a buffer of 2 MiB for the measurements, which is then shrunk down to 512 KiB, 128 KiB and 32 KiB, respectively, to increase the degree of contention, and thus the efforts required to maintain coherency. This shrinking of the buffer is shown along the Y-axis at the left side of Fig. 6. For each of these shared buffer sizes, we then perform 1024 accesses to random addresses in the buffer from both host , using single CPU core, and FPGA and track their latencies. As expected, contention increases both with a larger number of accesses, as well as with a shrinking size of the buffer. In both cases, the chance of a coherency conflict that has to be resolved increases. Interestingly, the additional coherency effort mostly affects the host's accesses, the latency of FPGA-side accesses stays almost constant. This is examined in greater detail on the right side of Fig. 6, which plots histograms of access times, now for 20,000 accesses, for the 32 KiB and 2 MiB shared buffer sizes. While being longer, the remote accesses from the FPGA-side have far less "jitter" (narrower distributions) than the local host-side accesses. Note that the very short outliers for the FPGA-side accesses are actually *hits* in the SC, for which the probability is larger in the smaller 32 KiB than in the larger shared buffer. In this experiment, only a single core on the host accesses the shared buffer. To investigate further, we used *multiple* cores on the host to modify and access the shared buffer. Our evaluation shows that increasing the number of cores from 1 to 3 for the 32 KiB address range actually *shrinks* the local host-side average access latency from 333 ns to 235 ns, due to more cache hits. On the other hand, the device access latency grows from 674 ns to 741 ns due to more cache misses. For the larger memory range, access times will again stay almost constant.

Algorithm 2: Pseudo-code for stressing coherency

```
Function coherency_stress_API (iterations)
    buffAddr = allocateBuffer(hugePage);
    for iterations do
        hostPtr = buffAddr + rand();
        FPGAPtr = buffAddr + rand();
        FPGA_Latency(start_CTG (FPGAPtr, length));
        host_Latency(write(dummyData, &hostPtr, length));
    end
end
```

Experiment 5: Atomic Operations. CCIX is also able to perform atomic transactions between RA (e.g., AU280) and HA (e.g., N1-SDP) by supporting AtomicStore, AtomicLoad, AtomicSwap, and AtomicCompare operations. They are constructed on the RA-side as multi-step sequences of AXI4-MM requests. Our evaluation shows that an AtomicCompare, initiated from the host, requires 50 ns, while one initiated from the accelerator takes 740-800 ns.

V. Database Application

After these detailed low-level measurements, we now examine the use of CCIX at the *application-level* for scenarios requiring fine-grained host-accelerator interaction. As a realistic scenario, we selected the domain of database acceleration. The studied system is neoDBMS (Fig. 7) [15], [16], a PostgreSQL-based DBMS using FPGA-accelerated NDP. In this manner, computations are moved closer to the storage (e.g., Flash, NVM), which is assumed to be directly connected to the accelerator. Using NDP reduces data transfers and increases overall system performance. However, NDP in database applications faces some challenges such as synchronization and transactional consistency. In the database, there are two types of transactions in NDP mode, *read-only NDP* and *update NDP*. In read-only NDP, to make transactions intervention-free, each

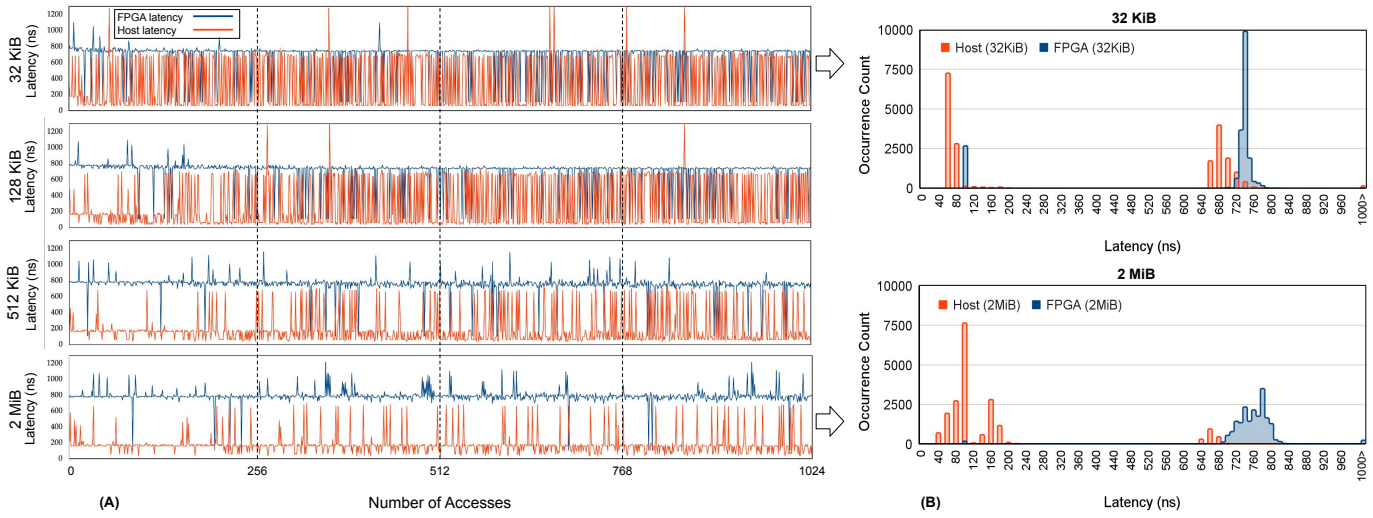

Fig. 6. Coherency efforts for increasing host-FPGA access contention using a single CPU core. Left (A): 1024 simultaneous random accesses in an address range shrinking from 2 MiB to 32 KiB. Right (B): Histograms showing "jitter" of access latencies for the two address ranges.

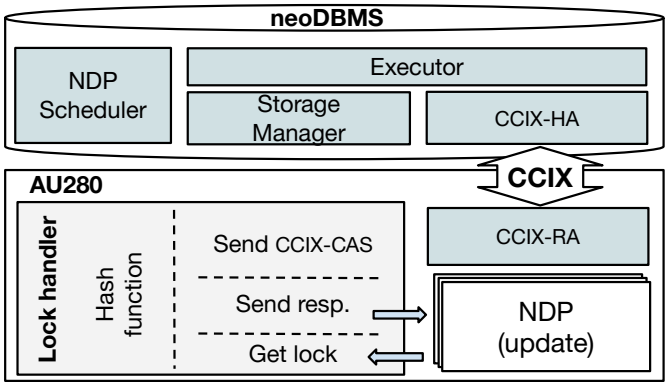

Fig. 7. neoDBMS architecture with shared lock-table

transaction operates against its own *snapshot*. This requires to first collect all DBMS updates in main-memory on the host, and then ship the changed DBMS state to the accelerator with each NDP-invocation [15].

In update NDP, making transactions intervention-free is challenging due to *concurrent* modifications from both the host and the accelerator to the same record. Initially, the same current version of the record is present on the accelerator and in the memory of the DBMS. If both simultaneously create a new successor version of the record, this results in *two* current version branches, causing unresolvable inconsistencies called a *write/write conflict*. One way to mitigate this inconsistency would be to exclusively lock the entire database table before execution, but this would severely limit concurrency. Another way is using a fine-grained cache-coherent shared lock table supporting record-level locking, so that each individual record's version can be locked to synchronize the modifications between DBMS and accelerator.

A. Shared Lock Table

To enable consistent and intervention-free update NDP operations between DBMS and accelerator, cache-coherent invalidation and synchronization mechanisms with low latency are necessary. To handle the *write/write conflicts* described above in neoDBMS, we realize a shared lock table by employing a CCIX-based solution. Without CCIX, synchronization would be far more costly and would most likely waste any performance gains achievable by NDP processing. To this end, our modified neoDBMS allocates a shared lock table in host memory and both parties, the host and the FPGA, request a lock on the record before updating it. neoDBMS relies on the huge page (i.e., HugeTLB Page) support in the Linux kernel to request physically contiguous memory pages used for the allocation of the lock table and makes sure they are pinned. As the size of the lock table is relatively small, and entries are very frequently accessed over the entire run-time of the DBMS, pinning the table in physical host memory is efficient.

Acquiring a row-level lock is performed by inserting an entry in a queue located in the hash bucket. The queue can thus contain multiple lock entries at the same time. The bucket position is calculated by applying a hash function to the record version identifier. Fig. 8 shows an example of two concurrent processes, one on the host and one on the device, requesting a lock of the same record version (i.e, R_{v2}). Applying the hash function on the record version identifier results in both processes trying to insert a lock into the same locking queue located in the same hash bucket, here numbered 2. In this example, first, the device requests a lock and immediately acquires the lock. The first slot represents the process currently holding the lock and which is allowed to modify the data. Later, the host tries to also request the same lock. Because the first slot of the lock queue is already taken, the host can not acquire the lock, and appends its request at the tail of the

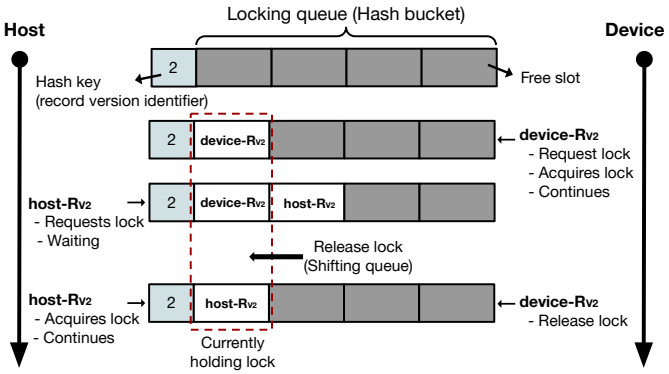

Fig. 8. An example for a single hash-bucket (for the hash key 2) in the shared-lock table, with concurrent lock requests from the host and the device to the same record version being queued in the bucket.

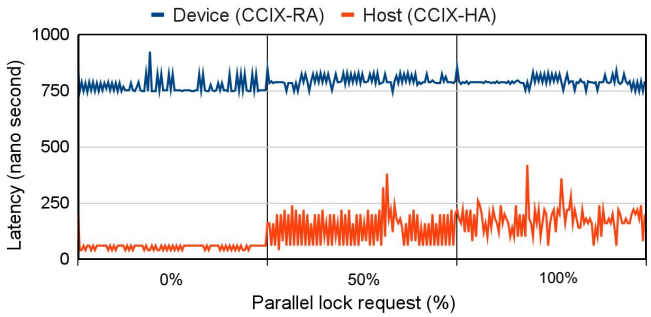

Fig. 9. The effect of parallel accesses to the shared-lock table

locking queue and waits. As soon as the device is finished, it releases the lock by *shifting* the entire queue left, granting the lock now located at the queue head to the next process. The lock is then acquired by the host and execution can continue.

On the FPGA, a Bluespec module has been developed to handle lock requests from the NDP-update module. This module creates a hash-table organized lock-table on the provided virtual address. The address of the allocated buffer and the lock table are specified by neoDBMS. The module receives/sends the lock requests/responses via streaming interfaces. After receiving a lock request, the module creates a CCIX Atomic Compare and Swap (CAS) operation to place the lock and update the queue, which the CCIX-RA on the AU280 then sends to the host. With the cache coherent shared-lock table and the employed CCIX atomic operations, we achieve a fine-grained cooperative processing of the data between DBMS and FPGA.

B. Evaluation

To evaluate the performance of the CCIX-based synchronization mechanism, we measured the end-to-end lock request latencies for neoDBMS running on the N1-SDP platform and the AU280-based accelerator as shown in Fig. 9. Since the size of the shared-lock table is larger than a Linux 4 KiB page, there is a high risk of accesses incurring long ATS overheads. This has been avoided here by using a huge page instead. The hardware module performs a single request independent of the actual shared lock operations to "warm-up" the ATC with a physical translation for the huge page. All actual lock requests will then have ATC hits and do not suffer from ATS overheads.

For the experiment, both neoDBMS (on a single CPU core) and the accelerator then continuously create lock requests, while we increase the contention on the other side. Under low contention, neoDBMS is able to lock a record version in 80 ns in the locally resident lock table. Under high contention, the local locking latency of neoDBMS increases to 200-250 ns. Locking from the accelerator of course takes longer, as remote accesses are performed to host memory, but the observed latencies of 750 to 800 ns are typical for CCIX atomic CAS operations (see **Experiment 5** above) and, most importantly, are *unaffected* by increasing contention. While this confirms the behavior already observed in **Experiment 4** above, it is interesting to note that it applies not just to the simple read/write operations of **Experiment 4**, but also to the more complex atomic CAS accesses used here.

VI. Conclusion

We have investigated the use of CCIX for fine-grained interaction between a host and FPGA-based accelerators. In our results we show, that especially for smaller transfer block sizes, shorter latencies are achievable than with PCIe. Furthermore, the transparent integration of address-translation with CCIX operations enables the cache-coherent Shared Virtual Memory (ccSVM) programming model between hosts and FPGA accelerators that has traditionally been the domain only of highly specialized platforms, such as the Convey HC-class machines. For the database use-case, it can be seen that CCIX remote accesses, while slower than local ones, do not suffer even for higher degrees of contested access to shared data structures such as the lock table.

As can also be seen from our results, optimization potential is present on multiple levels of the hardware/software stack. E.g., we have already demonstrated the use of huge pages to reduce address translation overhead. It would also be possible to insert more efficient application-specific translation mechanisms into the SoC, as all of translation occurs in the ATS Switch module, that, with its well documented interfaces, is amenable to be substituted with a custom version. This could be exploited, e.g., in the DBMS use-case of Sec. V to completely avoid ATS even for random access patterns exceeding the ATC capacities. There also appears to be optimization potential in the ATC itself, but this will require a larger engineering effort as it is more tightly integrated with the vendor-provided blackboxed part of the system.

Acknowledgment

The authors would like to thank Millind Mittal and Sunita Jain from AMD (Xilinx) Inc. for their support and the donations of hardware and software. This research was funded by the German Research Foundation (DFG) as project #419942270 *neoDBMS*.

REFERENCES

[1] D. de la Chevallerie, J. Korinth, and A. Koch, "fflink: A lightweight high-performance open-source pci express gen3 interface for reconfigurable accelerators," in *ACM SIGARCH Computer Architecture News*. ACM, 2015.

[2] *PCI Express Base Specification Revision 6.0*, PCI-SIG, January 2022.

[3] C. C. Inc., "An introduction to CCIX - white paper," 2016.

[4] D. Koenen and J. Defilippi, "CCIX: a new coherent multichip interconnect for accelerated use cases," http://www.armtechforum.com.cn/attached/article/C7_CCIX20171226161955.pdf, 2017.

[5] B. Benton, "Ccix, gen-z, opencapi: Overview & comparison," in *OpenFabrics Workshop*, 2017.

[6] D. D. Sharma, "Compute express link," *CXL Consortium White Paper.[Online]. Available: https://docs. wixstatic. com/ugd/0c1418 d9878707bbb7427786b70c3c91d5fbd1. pdf*, 2019.

[7] M. Slota, "Opencapi technology," in *OpenPOWER Summit*, 2018.

[8] *Gen-Z Core Specification 1.0*, Gen-Z Consortium, October 2018.

[9] P. Papaphilippou and W. Luk, "Accelerating database systems using fpgas: A survey," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 125–1255.

[10] M. Owaida, D. Sidler, K. Kara, and G. Alonso, "Centaur: A framework for hybrid cpu-fpga databases," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 211–218.

[11] *System Cache v5.0*, Xilinx Inc., November 2021, https://www.xilinx.com/support/documentation/ip_documentation/system_cache/v5_0/pg118-system-cache.pdf.

[12] *Versal ACAP CPM CCIX*, Xilinx Inc., November 2020, https://www.xilinx.com/support/documentation/architecture-manuals/am016-versal-cpm-ccix.pdf.

[13] *Versal ACAP Integrated Block for PCI Express v1.0*, Xilinx Inc., December 2021, https://www.xilinx.com/support/documentation/ip_documentation/pcie_versal/v1_0/pg343-pcie-versal.pdf.

[14] J. Korinth, J. Hofmann, C. Heinz, and A. Koch, "The TaPaSCo open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems," in *Applied Reconfigurable Computing*, 2019.

[15] A. Bernhardt, S. Tamimi, F. Stock, C. Heinz, C. K. Tobias Vinçon, A. Koch, and I. Petrov, "neodb: In-situ snapshots for multi-version dbms on native computational storage," *Proc. ICDE*, 2022.

[16] T. Vinçon, C. Knoedler, L. Solis-Vasquez, A. Bernhardt, S. Tamimi, L. Weber, F. Stock, A. Koch, and I. Petrov, "Near-data processing in database systems on native computational storage under htap workloads," *Proc. VLDB*, 2022.