CCKit: An Open-Source Toolkit for Cache Coherent Accelerators

ABISHEK RAMDAS, Systems Group, ETH Zürich, Zurich, Switzerland
DAVID COCK, Systems Group, ETH Zürich, Zurich, Switzerland
MICHAEL GIARDINO, Systems Group, ETH Zürich, Zürich, Switzerland
DARIO KOROLIJA, Systems Group, ETH Zürich, Zurich, Switzerland
ANASTASIIA RUZHANSKAIA, Systems Group, ETH Zürich, Zurich, Switzerland
DANIEL SCHWYN, Systems Group, ETH Zürich, Zurich, Switzerland
ADAM TUROWSKI, Systems Group, ETH Zürich, Zurich, Switzerland
GUSTAVO ALONSO, Systems Group, ETH Zürich, Zurich, Switzerland
TIMOTHY ROSCOE, Systems Group, ETH Zürich, Zurich, Switzerland

The trend toward system specialization is leading to a proliferation of accelerators, exposing interconnects as serious bottlenecks, both in functionality and performance. As a result, several alternative approaches have been proposed which promise to expand the coherence domain beyond homogeneous sockets to rack scale heterogeneous systems. In parallel, GPU vendors have developed their own high bandwidth interconnects also aiming for heterogeneous coherence beyond the CPU. This expansion of the coherency domain raises many questions that remain unanswered, in particular, how devices other than CPUs will interact with the coherence protocol and whether applications can take advantage of these expanded domains. As protocols such as CXL are still evolving, it is important to explore alternative designs that go beyond what the commercial specifications dictate. For this purpose, we developed CCKit, an open-source, server-class toolkit comprising a complete cache coherency stack on reconfigurable accelerators. CCKit is more flexible than commercial products and its performance is highly competitive with hardware-based implementations, thus enabling important and novel application use-cases for expanded coherence domains. Experimental data from real workloads provide the ability to influence and expand future interconnects, protocols, and applications.

CCS Concepts: • **Computer systems organization** → **Reconfigurable computing**; **Heterogeneous (hybrid) systems**; • **General and reference** → *Experimentation*; • **Hardware** → *Buses and high-speed links*; **Reconfigurable logic applications;**

D. Cock is currently with Neutrality.
M. Giardino is currently with the Computing Systems Lab Huawei Technologies Co. Ltd, Zürich, Switzerland.
D. Korolija is currently with Advanced Micro Devices, Switzerland.
D. Schwyn is currently with Neutrality.
Authors' Contact Information: Abishek Ramdas, Systems Group, ETH Zürich, Zurich, Zürich, Switzerland; e-mail: ramdas.abishek@proton.me; David Cock, Systems Group, ETH Zürich, Zurich, Zürich, Switzerland; e-mail: david.cock@inf.ethz.ch; Michael Giardino, Systems Group, ETH Zürich, Zürich, ZH, Switzerland; e-mail: michael.giardino@huawei.com; Dario Korolija, Systems Group, ETH Zürich, Zurich, Zürich, Switzerland; e-mail: dario.korolija@inf.ethz.ch; Anastasiia Ruzhanskaia, Systems Group, ETH Zürich, Zurich, Zürich, Switzerland; e-mail: anastasiia.ruzhanskaia@inf.ethz.ch; Daniel Schwyn, Systems Group, ETH Zürich, Zurich, Zürich, Switzerland; e-mail: daniel.schwyn@inf.ethz.ch; Adam Turowski, Systems Group, ETH Zürich, Zurich, Zürich, Switzerland; e-mail: adam.turowski@inf.ethz.ch; Gustavo Alonso, Systems Group, ETH Zürich, Zurich, Zürich, Switzerland; e-mail: alonso@inf.ethz.ch; Timothy Roscoe, Systems Group, ETH Zürich, Zurich, Zürich, Switzerland; e-mail: troscoe@inf.ethz.ch.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).
ACM 0734-2071/2025/09-ART9
https://doi.org/10.1145/3763790

Additional Key Words and Phrases: Cache coherence, reconfigurable computing, disaggregated systems, far memory, near memory compute, operator pushdown, interconnects

ACM Reference Format:
Abishek Ramdas, David Cock, Michael Giardino, Dario Korolija, Anastasiia Ruzhanskaia, Daniel Schwyn, Adam Turowski, Gustavo Alonso, and Timothy Roscoe. 2025. CCKit: An Open-Source Toolkit for Cache Coherent Accelerators. *ACM Trans. Comput. Syst.* 43, 3, Article 9 (September 2025), 30 pages. https://doi.org/10.1145/3763790

1 Introduction

There is increasing interest in extending cache coherence, long regarded as essential for parallel programming on homogeneous multiprocessors, to other parts of a computer system and in opening up hardware coherence protocols for other uses. The main trend driving this renewed interest in coherence protocols is the rise of heterogeneous hardware in the form of Systems-on-Chip (SoCs), and accelerators such as GPUs, FPGAs, TPUs, and so on. Such a shift in hardware design is in turn driven by performance scaling [24, 39], parallel machine learning workloads [30], and specialization [98]. When a computer is a collection of heterogeneous processing elements of equal standing, the question arises as to how much of the system should be coherent.

The proliferation of accelerators has also driven innovation in the interconnects linking them to the CPU. Because PCIe lacks the necessary features to support increasingly sophisticated and powerful accelerators, proposals like CCIX [25], GenZ [42], and OpenCAPI [93] emerged. Many of these have converged into Compute eXpress Link (CXL) [34], whose family of standards has emerged as the front-runner in CPU-centered systems. However, there are other, more specialized standards for GPUs [86] (NVIDIA NVLink [40, 65], AMD's Infinity Fabric [10]) that offer a different set of features from CXL. Recent developments [101] suggest that interoperability between competing protocols using sub- or supersets of features is on the horizon, but the details remain murky. Additionally, there is an entire ecosystem of interconnects and protocols for RISC-V and embedded systems [11, 104]. Interestingly, all these efforts provide cache coherence and/or coherent memory access in ways unavailable before. While traditional coherence used proprietary interconnects between parts from a single vendor, it is now closer in spirit to network protocols (see, e.g., [66]).

The generality and flexibility of these interconnects enable innovative architectural designs exploiting coherence, such as disaggregated memory [22] or crash consistency for persistent memory [12, 16]. Some even argue that cache coherence protocols should be tailored to the application [71, 104, 116] rather than offered as a black box. However, this requires the tools and sufficiently high-level interfaces to allow applications to interact with the hardware cache protocol. Working with coherence protocols, even those designed with interoperability in mind, is highly challenging. Real coherence protocols are complex, with hundreds of transient states and many potential race conditions [73]. Implementing a coherent endpoint as part of an application is difficult and time-consuming [18, 78]. Reusing an implementation is even harder, particularly when the protocol is being used non-traditionally.

Simulation fares poorly in these scenarios: either the simulator is painfully slow, making it hard to derive meaningful results in the presence of I/O and real-world interactions, or it achieves better performance by simplifying the protocol, potentially losing critical, practical issues [18, 76, 86].

To address this, we present CCKit, an open-source, server-grade, modular, and flexible coherence protocol design and implementation. We focus on FPGAs as their reconfigurablity is ideal for exploring the design space and meeting the performance requirements of low-level CPU interaction [48, 85]. Indeed, many proposals taking advantage of coherent interconnects are FPGA-based [12, 22, 23, 60] and companies are already patenting use-cases based on cache coherent FPGAs [20, 21]. FPGAs are also a standard component in the cloud (Microsoft [70, 81], Amazon [6], Alibaba

[28]), with novel applications e.g., acceleration of database engines [79, 114] that would greatly benefit from using coherent FPGAs.

Prototyping with CCKit is fast and faithful: its first implementation runs natively on a real hardware platform [31], and includes a performant coherence implementation matching the speed of the CPU. CCKit is also flexible: it exposes to applications much more about protocol events than emerging standards do. Crucially, modifications are simplified by abstracting most of the state machine complexity of the coherence protocol while exposing enough low-level access to allow a wide range of use-cases. To ensure flexibility, CCKit is built as an intermediate layer between the raw coherency messages delivered from the interconnect and the application logic and offers high level and well-defined interfaces, making it portable to future standards providing symmetric coherence such as CXL 3.0. CCKit is not intended to compete with commercial interconnects, but to provide a vehicle to build symmetric cache coherent accelerators and applications before products are available. Moreover, with a wide spectrum of design choices and architectures available, the flexibility and extendability allows for proposing future interconnect standards by: (1) determining what is needed on the accelerator side to implement cache coherency; (2) explore applications and software architectures that can take advantage of cache coherency; and (3) identify performance and design pitfalls arising from coherence protocols that might not be suitable to common use cases.

We show the performance and versatility of CCKit through micro-benchmarks and several acceleration use cases. The former demonstrate that CCKit on an FPGA has performance comparable to a CPU, despite the lower clock frequency on the FPGA. Our use cases explore (a) the implementation of a custom pre-fetcher on the FPGA (doubling the read throughput from FPGA memory from 7.8 GiB/s to 17.4 GiB/s); (b) the maintenance of database views with update propagation from base tables to an aggregated view (running at interconnect speed of 19.5 GiB/s); and (c) synchronous RPC from CPU to Field Programmable Gate Array (FPGA) based on the CCKit directory controller, outperforming both programmed I/O and DMA (null RPC P_{50} latency of 900 ns).

In this article, we present the following contributions:

— CCKit: a fully open, modular, and symmetric coherence protocol design that provides a generalized interface to applications
— A flexible programmable architecture for cache coherence on an FPGA
— A performant implementation of CCKit on a server-grade platform that allows for the development, acceleration, and evaluation of complex enterprise workloads
— A toolchain to generate correct application-specific protocols
— Evaluation of CCKit across several workloads: microbenchmarks, a custom memory prefetcher, maintainance of database views using coherence protocol, lockless coherent table updates, and a synchronous CPU-FPGA RPC

The remainder of the article is presented as follows: Section 2 presents the current field of interconnects and coherence protocols and the placement of CCKit in this context. In Section 3 we present the approach and high-level design of CCKit and the interfaces. The implementation details are presented in Section 4 including techniques for obtaining CPU-like performance on an FPGA. The microbenchmarks and application use cases are presented in Section 5. Additional related work can be found in Section 6 before concluding in Section 7.

2 Background

In this section we provide background into types of coherent protocols (Section 2.1), the types of interconnects and models of coherence in heterogeneous systems (Section 2.2) and the varied forms of coherence in Multiprocessor System-on-chips (MPSoCs). This motivates the primary question posed by this research—what models of coherence are appropriate in modern large-scale heterogeneous systems? In order to answer this question, we must first determine what the design and interfaces (Section 3) and, by extension, implementation (Section 4) of CCKit will be.

2.1 Symmetric vs. Asymmetric Protocols

A crucial aspect in cache coherence protocols is who controls the protocol, with models of cache coherence broadly falling into two categories: *asymmetric* and *symmetric* [74].

Asymmetric protocols preserve the host-device relationship between CPU and accelerator: both sides can implement *caching agents* (and cache data), but only the CPU implements a *home agent* which tracks ownership of cache lines. This simplifies accelerator implementation but limits scalability and flexibility. It also significantly affects performance: to access local memory shared with the CPU, an accelerator must make a request to the CPU. In this model, the accelerator's data is, from the start, *a copy* and there is no notion of the accelerator ownership of data that the CPU might cache. Some of the asymmetric examples that are mentioned in Chapter 10 of [74] are the CPU-GPU heterogeneous protocol with selective caching [3] or architectures with a global directory residing on the CPU side and *stitching* together different cache coherence protocols on CPU and GPU nodes.

Moreover, the FPGA loses control over an entire section of the coherence protocol. For example, applications cannot issue messages that belong to the directory protocol such as the forward downgrade messages and might be limited to load and store operations performed through the caching agent. Finally, the FPGA also loses observability over the coherence protocol. For example, applications on the FPGA cannot observe any messages that are issued by other coherence controllers to the home agent for its own memory. With the concurrency and dynamism of coherence protocols, not being able to access the global view of the states of cache lines reduces the guarantees that can be inferred by applications, making it difficult to design applications that can be proved to be correct.

Symmetric protocols have home agents on both CPU and accelerator, as in a homogeneous NUMA system. While more complex to implement, they provide seamless coherent integration between the CPU and accelerator. Less obviously, they allow the accelerator to unconventionally participate in the protocol. Rather than simply observing transactions on the CPU cache and being notified by the CPU, the accelerator can actively generate its own notifications and manage its own memory independently. In [74] under symmetric case fall the distributed directory-based protocols covered in Chapter 8.

2.2 Accelerators and Coherent Interconnects

Until recently, accelerators like GPUs used a "host-device" computational model based on PCIe in which the host CPU manages external accelerator resources. Data is offloaded in bulk for processing and the results copied back to the host. This is the model used by CUDA [75], OpenCL [91], and modern accelerators such as TPUs [55] and VCUs [83]. It arose in part from the lack of cache coherence between host and device, and favors highly structured workloads that can be expressed as offloaded batches. This model implicitly assumes the accelerator takes a copy of the input data, performs a task, and returns results without engaging in any complex exchange or interaction with the CPU [77].

As accelerators have become more powerful (in some cases, many CPUs are needed to feed a single accelerator [115]) PCIe standards have greatly increased bandwidth. However, the underlying principles of PCIe remained unchanged, despite its limitations in terms of protocol and features, and the diverse proliferation of accelerators.

Intel HARPv2 [77] and IBM CAPI [92] were early attempts at better accelerator integration, coherently connecting a server-class CPU and FPGA. HARPv2 used an *asymmetric* implementation of the symmetric QPI protocol [33] (in contrast to other approaches available [102]), while CAPI used a PCIe Host Bridge and Coherent Accelerator Processor Proxy on the CPU, and a service layer on the FPGA. In both cases the protocol is asymmetric and closed: the FPGA application has little access and control over the coherency protocol.

Later developments include CCIX [25] which supports a symmetric protocol by extending PCIe, and OpenCAPI [93] which implements an asymmetric protocol over PCIe and Bluelink. Both require accelerators to work with caching enabled and use virtual addresses, translated by the CPU's MMU. Performance studies of CCIX-attached FPGAs [94] have emphasized the importance of cache coherence in heterogeneous architectures.

CXL [34] builds coherence and memory semantics on top of PCIe and provides a unified coherent memory space between CPU and accelerators. Currently, the first CXL 1.1 hardware is becoming available, using an asymmetric protocol with coherence bypass for direct access to unshared device memory. Symmetric coherence is planned for CXL 3.0 [89], which has triggered interesting ideas around what it will allow, based on simulations that promise impressive throughput and latency, as well as extensibility beyond one machine [54, 66, 69]. CXL retains a somewhat prescriptive position on the use of cache coherence messages that favor fine-grained acceleration idioms, like work stealing, to accelerate applications. More importantly, it is not clear what the appropriate *interface* between accelerator logic and a complex protocol like CXL should be. It is this latter question that CCKit addresses.

While cache-coherent accelerators and GPUs are becoming available, applications which take advantage of coherence beyond straightforward unified address spaces are still rare. We attribute this to the limited availability of hardware allowing for such specialized and application-driven protocol (ab)use. Such hardware is in turn rare as manufacturers have a limited incentive to produce and market it without demonstrated industrial applications.

We have developed CCKit, on the Enzian research platform, in an attempt to break this chicken-and-egg cycle. Our hope is that by providing a realistic, well-supported platform for research into unconventional applications of coherence we can provide application developers with an accessible path to explore these ideas, and to collect the necessary evidence to motivate the further development of such flexible coherence platforms.

While we have attempted to select several realistic applications as illustrative examples in this article, such as database acceleration (Section 5.6) and remote procedure calls (Section 5.7), we cannot hope to fully predict the scope of potential applications. CCKit is merely a first step: pointing to a fruitful area of investigation, and providing a solid foundation on which to build.

2.3 Coherence in MPSoCs

Alongside these developments in server architecture, processors have evolved toward less homogeneous and more eclectic designs formed around heterogeneous SoCs and chiplets.

The most mature and broadly adopted coherent CPU-FPGA systems combine both on a single MPSoC, such as Xilinx Zynq UltraScale+ [107] and Intel Agelix [52]. Sharing an SoC simplifies the physical interconnect and provides both coherent and non-coherent ports between CPU and FPGA. Coherent access is generally asymmetric: the FPGA can access the CPU's LLC. This tightly-engineered integration significantly limits both application flexibility and available CPU performance, the norm being simple dual or quad-core ARM processors aimed at embedded systems. More flexible but tightly coupled CPU-FPGA systems have been proposed [64], but their availability and programmability is unclear.

RISC-V's TileLink takes a more general approach [11, 32, 96] aimed at low-latency connectivity between CPUs, caches, accelerators, memory, and other SoC components. TileLink has a number of coherence *policies* which can be subsets of the MOESI protocol. Multi-socket coherence can be achieved with OmniXtend [82]. While implementations for high-end CPUs and accelerators (including FPGAs) have yet to appear, TileLink shows a clear response to the demand for customizable coherent interconnects in increasingly heterogeneous systems, a demand also observed by others [18, 49, 71, 116].

As mentioned, implementing correct coherence protocols and controllers is a complex undertaking. Bring-your-own-core [8] simplifies the generation of protocols for integration of heterogeneous devices, allowing for much of the complexity to be hidden, while work like Crossing Guard [78] and Spandex [5] integrate disparate accelerators through an intermediate interfaces. Several recent research efforts propose the design or generation of coherent controllers [18], NoCs [38, 46], and entire SoCs [8, 41, 45, 68, 80, 100], including CPU-FPGAs SoCs [64]. OpenPiton [7], BlackParrot [80], and others [36, 41, 104] have been taped out. BedRock [104], built with BlackParrot [80], comes the closest to CCKit in creating an entire configurable coherence stack, albeit at the cost of custom RISC-V extensions and no virtual memory or interrupts. It provides correct implementations within the bounds of standard coherence protocols (e.g., MOESI, MSI, MESIF), but it's goal is to work at the lowest level of coherence, as these systems are primarily aimed at SoCs. Sensibly, the focus is RISC-V, considering the open nature of the ISA, protocols, and many available designs. Systems like Cohort [100] acknowledge the need for better accelerator integration beyond the SoC level, however this remains unimplemented. None possess a modern, server-grade CPU, but more importantly for our target applications, they do not have the necessary system support for distributed, large-scale datacenter workloads.

3 Approach and Design

CCKit allows FPGA applications to interact directly with a cache coherence protocol in a more flexible way than assumed by simple coherence. It abstracts away most protocol complexity, providing a portable interface for application logic to coherently access memory alongside the CPU and also, crucially, to interact with the CPU's LLC.

CCKit consists of a hardware component which can be instantiated on an FPGA, and provides the simplified interface to user logic, and an OS kernel module which allows access to CCKit from CPU software. Key to CCKit is factoring the coherence protocol into scalable re-usable hardware units.

The design and interfaces of CCKit are applicable to a range of hardware platforms and coherence protocols, although an implementation will be specific to a particular CPU, protocol, and platform. In this section, we describe the general architecture of CCKit, and in Section 4 discuss our first implementation using the Enzian platform [31]. To the best of our knowledge, CCKit on Enzian is the only implementation of a fully symmetric coherent CPU-FPGA platform.

3.1 Target Platforms and Assumptions

CCKit makes fairly relaxed assumptions about the underlying hardware. We target 2-node systems where one node is a conventional multicore CPU, and the other is an FPGA and a MESI-based directory-based write-invalidate cache coherence protocol connects the two. Physical address space is partitioned between the two nodes. CCKit assumes an architecture-specific layer on the FPGA which exchanges messages with the CPU, guaranteeing delivery and deadlock-freedom but not ordering.

These assumptions are reasonable: most modern coherent multi-socket systems adopt a directory-based [26, 95] write-invalidate MESI-like protocol rather than less-scalable, broadcast-based *snooping* [84] protocols. Moreover, the complex network topologies envisioned by Cache Coherent Interconnect for Accelerators (CCIX) and CXL 3.0 are orthogonal to the goal of CCKit, which is to provide a clear but rich high-level interface between accelerator logic and the coherence protocol. This allows the underlying state machines implementing the coherence protocol to change for different topologies, but CCKit's architecture and interface remain the same.

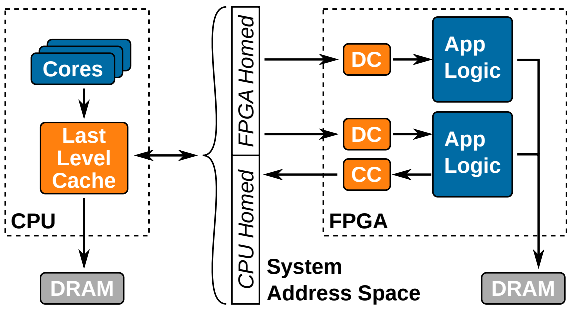

Fig. 1. The CCKit architecture consists of a set of DCs and CCs connected to arbitrary application logic on the FPGA which in turn connected to DRAM. The DCs and CCs interact over ECI with the CPU via the LLC.

The key challenge that CCKit addresses is this: in practice, race conditions, message reordering by the interconnect, and concurrency mean that real implementations have many more hidden, intermediate states than the textbook MESI states, greatly complicating the protocol. More than 100 states is not unusual in a multi-socket system. The ability of FPGAs to handle this complexity while operating at a lower frequency is cited as an argument for using asymmetric protocols or no coherency at all in CPU-FPGA systems [35]. CCKit refutes this argument, providing a full symmetric protocol implementation that (as we show in Section 5.2) keeps pace with the native CPU implementation while exposing a richer interface.

3.2 High-Level Architecture

Figure 1 shows the architecture of CCKit. We treat cache lines homed on the FPGA and the CPU differently. A Directory Controller (DC) component maintains the directory information for FPGA-homed lines, including the local protocol state and the state it believes the line to be in on the CPU. CPU-homed lines are handled similarly by a Cache Controller (CC) component on the FPGA. The implementation and interface is similar to the DC, but since it does not need to maintain the directory the CC is rather simpler. We concentrate on the DC in this article; our applications in Section 5 use only the DC.

Note that in neither case does CCKit actually cache the line itself – this is left as a choice to the application logic. Whether FPGA-based caches are beneficial is an open question, and applications presented here directly manipulate data in memory.

Each DC or CC is responsible for a disjoint region of physical address space. By varying the number of units, performance can be traded off against FPGA resources. This mirrors the behavior of a CPU LLC, except that the CPU controllers' parameters are hardwired (the only "application" to be supported is the cache itself).

CCKit's architecture is layered. The coherence protocol implementation at the lowest layer simply delivers messages from the coherent link. The DC and CC sit above and receive the coherence messages. The application logic lies at the top and implements the accelerators functionality. These components handle coherence transactions and maintains coherence invariants at cache line granularity. Such transactions can be initiated remotely by the CPU or locally by application logic, and transactions on different lines are handled entirely independently. In fact, though application logic, shown in Section 5, we can create dependencies between them.

3.3 FPGA-Side Interface

A key feature of CCKit is the interface between the DC and user logic on the FPGA with a focus on providing maximum controllability and observability of the coherence protocol to user logic. Much

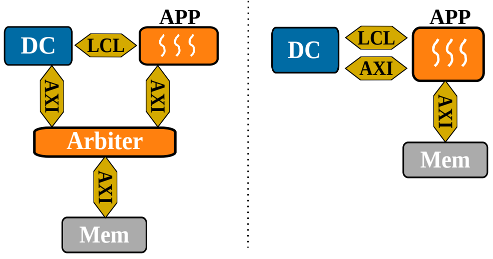

Fig. 2. We see two possible configurations of a user application using CCKit. The access to memory, AXI, and LCL channels can be arbitrarily configured.

of the complexity of a directory-based cache coherence protocol is due to concurrency: different nodes can issue operations on the same cache line, and coherence messages can be in-flight or be re-ordered by the interconnect. The DC (and CC) hide the complex state machine required on the FPGA to maintain coherence invariants and track the state of each line on the other (CPU) node. The DC instead exposes to application logic on the FPGA a simplified state machine reflecting the familiar MESI states. The user logic does not have to track detailed cache line states but can rely on the guarantees provided by DC. This abstraction is generic enough to cover a range of underlying MESI-like coherence protocols, but at the same time richer than conventional coherent memory.

Specifically, the DC exposes to application logic all transitions between the primary stable states of the cache line on both the FPGA and CPU (as far as this can be determined from protocol messages). This is in addition to seeing reads and writes by the CPU to the cache line, and supporting reads and writes from application logic on the FPGA. Finally, user logic can issue requests to the CPU's LLC which are a superset of the usual *clean* and *invalidate* operations.

This is provided, as shown in Figure 2 via an AXI interface for reads and writes (to service upgrade and downgrade requests, respectively), plus a request-acknowledge interface allowing FPGA logic to trigger, e.g., *clean* (write-back without invalidate) or *clean+invalidate* operations in the CPU's LLC, and finally an AXI-lite bus for I/O configuration.

In addition, FPGA logic can *lock* cache lines from being upgraded by the CPU upon completion of the clean or clean-invalidate operation, ensuring that the CPU's LLC cannot hold a modified copy of the cache line and the FPGA memory must hold the most up-to-date copy until it unlocks the line.

In the simplest use-case, the AXI interface can connect directly to FPGA DRAM controllers to provide coherent access to FPGA-side DRAM from the CPU. However, non-trivial applications instantiate their own logic between these components to interact with the coherent interconnect (Section 5). In Figure 2, two possible configurations of the app relative to the DRAM and DC are shown.

This interface, while relatively protocol-independent, is sufficient for many useful interactions with the CPU's LLC. For example, an application observing an AXI read request for a cache line will infer that the line is invalid in the LLC and the FPGA has the most up-to-date copy. Similarly, an AXI write request indicates that the line was either *invalid* or *shared* in the CPU's cache but never *exclusive* or *modified*.

Applications which additionally use the request-acknowledge interface (e.g., Section 5.6, Section 5.4) can issue *clean*, *clean-invalidate*, *lock*, and *unlock* (*LCL* interface in Figure 2) requests for cache lines, providing full flexibility in managing the coherence protocol. A state change to a locked state is currently only possible through a clean message, unlock can be done explicitly. For example, the algorithm to perform a clean action on the cache line, that is in some state on the CPU, would be to issue a clean request, which will lock the cache line, send invalidate message to the CPU, wait for the CPU acknowledgment with or without data, wait for a completion of a potential write, acknowledge the clean operation, and unlock the cache line.

In total, the application can observe read requests and return the data back, observe write requests and store data to the memory, generate clean messages, thus forcing the up-to-date content of a cache line to be returned to the device, and unlock (lock) a cache line in order to prevent CPU's updates. For example, a user can develop application-specific coherence protocols on top of the coherence protocol layer, which can be used to maintain application-specific invariants between unrelated cache lines (e.g., Section 5.6).

3.4 CPU-Side Interface

In operation, the CCKit interface to software is relatively simple as coherence is mostly transparent to software, and what explicit cache operations the CPU supports (flush, invalidate, etc.) simply translate into coherence messages. Any further coordination, such as using particular cache lines to synchronize a protocol (as demonstrated in Sections 5.6 and 5.7), is application-specific and fully visible to software.

We do not, in this article, explore the full challenge of integrating special-purpose and heterogeneous memory management in a general-purpose operating system. Linux does have support for a form of heterogeneous memory management, however we were unable to use existing kernel mechanisms to manage CCKit's memory. We have added sufficient mechanisms to Linux, in the form of a loadable kernel module, to allow user-space processes to map DC-managed FPGA-side addresses with the necessary attributes. This is sufficient for the evaluation reported here, but its design is deliberately kept as simple as possible, and does not attempt to manage these resources in a coherent manner.

The kernel module represents the FPGA's memory space as a device file, in the manner of `/dev/mem`. The module implements the `mmap()` syscall, allowing a process with the appropriate permissions to map any desired sub-region of the FPGA address space into its own virtual address space. While the necessary functionality is straightforwardly supported by the MMU, certain assumptions in the Linux memory management implementation introduce a few challenges.

For example, huge page mappings reduce TLB pressure and page-table footprint. The applications we present here also derive no benefit from fine-grained (e.g., 4 KiB) mappings. Also, as these addresses do not actually represent pageable memory, it is generally preferable for the page tables to be populated eagerly. One instance where this is significant is where a CCKit protocol depends on prefetch hints: the CPU cache silently drops these where they would trigger a page fault.

The CCKit kernel module bypasses the standard in-kernel interfaces for page mapping. This is a neither a limitation of CCKit nor of Linux, but reflects the fact that heterogeneous devices such as this are using physical addresses in ways quite different to what a conventional system assumes. We do not address this in this work, but recognize that it is a topic of great interest as heterogeneous systems become more mainstream, that we have ourselves investigated in other work. We have previously presented [2] an approach using an extension of the Barrelfish [9] research operating system's capability model to securely manage arbitrary physically-addressible resources between components with differing views of the system's address spaces.

4 Implementation Details

Our first implementation of CCKit is on the publicly available Enzian computer [31, 97], a 2-socket heterogeneous server platform. One socket holds a Marvell ThunderX-1 CN8890-NT 48-core ARMv8-A CPU running at 2.0 GHz. It has a 2-level cache with a 16 MiB shared LLC, using 128 B lines, connected to 4 × 32 GiB 2133MT/s DIMMs. The other socket contains a Xilinx VU9P UltraScale+

FPGA [105], with (in our case) 4 × 16GiB 2400MT/s DIMMs. The CPU's native interconnect is exposed to user FPGA logic as the Enzian Coherent Interconnect (ECI); this inter-socket link has a theoretical bandwidth of 30 GiB/s. About 20 GiB/s is achievable in practice with a round-trip latency of 230 ns (Section 5.2). Two DDR4 channels (on either node) are sufficient to saturate this link. We build on the existing open-source FPGA "shell" for Enzian, which exposes raw inter-socket protocol ECI messages to the FPGA. Above this, we implement the DC and CC.

4.1 Underlying Hardware Coherence Protocol

Implementing CCKit on Enzian entails (1) interfacing to raw ECI messages so that the FPGA appears to the CPU as NUMA-remote coherent memory, and (2) providing the CCKit interfaces to the user's FPGA application logic.

ECI is a fully symmetric coherence protocol. While it carries other traffic (e.g., I/O, interrupts, atomics), we are concerned with the coherent memory interface, exposed as 5 flow-controlled, reliable virtual circuits in each direction, each devoted to a message type: *request with data*, *request without data*, *response with data*, *response without data*, and *forward without data*. For increased parallelism, all channels are divided in two by the ThunderX-1, each handling either even- or odd-numbered lines.

The key messages are *Read Shared/Exclusive*, *Upgrade to Exclusive*, *Voluntary Downgrade Dirty/Clean*, and *Forward Shared/Exclusive*, and their matching responses. *Read*, *Upgrade*, and *Voluntary Downgrade* requests are sent from the remote node to the home node, while *Forward* requests are sent by the home node to force the remote node to write back. To these platform-defined channels we add local operations carrying user requests like force invalidation or lock/unlock a line. Requests and responses on this channel are treated exactly as those on the ECI channels.

Above this, CCKit implements an efficient, deadlock-free, and scalable design providing access to the full address space while maintaining coherence invariants. There are two key implementation challenges: first, how to ensure that the protocol state machine is correct with regard to the processor's implementation, and second, how to saturate the achievable 20 GiB/s of the ECI link with acceptable FPGA resource consumption.

4.2 Correct Protocol State Machine Generation

Implementing CCKit requires the full coherence protocol to be specified and the DC state machine to be correctly implemented. Here we provide an overview description of this process. Coherence protocols require the maintenance of two invariants: single-writer-multiple-reader and data-value. In the scope of CCKit, the CPU is allowed to perform both caching and non-caching operations but applications on the FPGA are restricted to perform only coherent non-caching operations. This was done to simplify both the implemented protocol and the applications themselves for evaluation. Since the CPU is the only entity that can have a modified copy of a cache line, the single-writer-multiple-reader invariant is maintained by default.

The CCPI specification provides the list of coherence messages and information about common types of transactions. We also collected traces of traffic between two CPUs in order to observe the behavior of a correctly implemented system. From these, we developed an abstract model of the system with the CPU, FPGA-DC, and interconnect as entities. We also identified rules of interaction between them when only a single cache line is involved.

The first step in generating a specification is to identify the rules of interaction between the CPU and the DC in the coherence protocol. The rules are as follows: First the CPU initiates a non-posted *upgrade* transaction by issuing an upgrade request and waiting for a response from the DC. The upgrade request can be from any lower state to any higher state (*I to S, I to E, or S to E*). The DC is responsible for maintaining the coherence invariants (by keeping track of home and remote states of the cache line in its directory) and responding to the request with an acknowledgment. Second, the CPU can issue a posted *voluntary downgrade* transaction to downgrade from any higher state to any lower state (*E/M to S, E/M to I, S to I*). Third, the DC can initiate a non-posted *forward downgrade* transaction to request the CPU to downgrade from any higher state to any lower state. The CPU is responsible for downgrading the state of the cache line and sending an acknowledgment. Fourth, a transaction that has been initiated cannot be canceled and the protocol does not allow for negative acknowledgments. Finally, this protocol uses timeouts to identify failures.

Next we look at the nature of the interconnect and how it affects coherence traffic. First, the interconnect is guaranteed to be deadlock free with separate virtual channel for different message-classes and non-interfering flow control. Second, the interconnect is reliable and guarantees delivery of messages: A sender does not have to issue the same message multiple times. Third, the interconnect does not guarantee any ordering between coherence messages: If there are *n* messages in transit, they can be received by the receiver in any of *n!* ways. Fourth, the interconnect as a non-zero latency and delivery of messages is not instantaneous.

Most of the complexity of the coherence protocol is to handle *conflicts* that arise due to two reasons: the latency of the interconnect and the reordering of coherence messages by the interconnect. To illustrate how latency of interconnect can cause a conflict, consider the CPU evicting a cache line from its Last-Level Cache (LLC) by posting a voluntary downgrade message. While the message is in transit, the DC can issue a forward downgrade request which will be received by the CPU that does not have a copy of the cache line. Such conflict scenarios are handled natively by the protocol through a special class of coherence messages called *conflict responses*. Next, to illustrate how reordering can cause conflict, consider the scenario where, for a cache line, the CPU downgrades it and immediately follows up with an upgrade request. If the messages are received out-of-order by the DC, it has to contend with the conflict of CPU requesting a cache line that should already have been cached in the CPU. The DC handles such conflicts by having intermediate states to keep track of out-of-order and in-flight messages.

Using these rules of interaction, we can then build a model of the coherence protocol that can be used to enumerate all possible coherence interactions that would have to be handled by the DC. We start by considering only the subset of interactions where there are no conflicts. Given the initial state of the cache line in the CPU, we can enumerate all possible pathways the interactions can take. For example, if the CPU has a cache line in *invalid* state, there are only three possible pathways: The CPU continues to have the cache line in *invalid* state, or the CPU issues an upgrade request to *shared* state (and wait for DC's response), or the CPU issues an upgrade request to *exclusive*. Each pathway is a coherence transaction and can be represented in the form of a *state-equation* which contains the initial home and remote state of the cache line in DC's directory, the sequence of messages received by the DC, the final home and remote state of the cache line, and the action to be performed by the DC.

Subsequently we broaden the set of coherence transactions by identifying conflict scenarios that arise due to latency of interconnect. Given the model and rules of interaction, we can enumerate all possible conflict coherence transactions and identify the action to be performed by the DC in each scenario. These coherence transactions are also specified as state-equations. Finally we account for reordering by interconnect by taking existing state equations with *n* (>1) messages and creating *n!* new state-equations. This completes the set of state-equations required to formally specify the coherence protocol.

CCKit uses state space exploration offline to generate a state machine with all possible intermediate states from this specification, ensuring that coherence invariants (SWMR and data-value invariant [74]) and deadlock freedom are maintained, and optimizing for performance. Building a state machine for single-message coherence transactions is trivial and requires only one transition

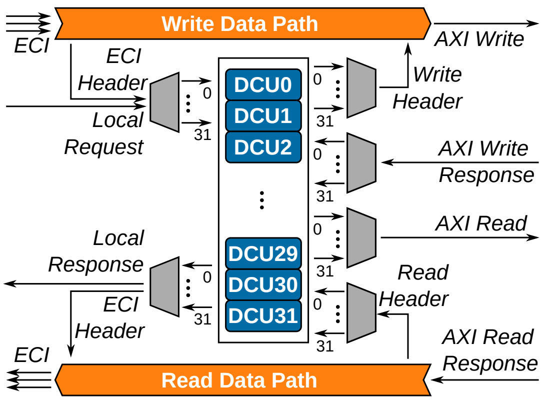

Fig. 3. The Directory Controller Slice (DCS) contains several individual DCUs. This figure shows the read and write responses between the AXI bus and ECI.

rule. As the cardinality (number of messages) of a coherence transaction increases, CCKit must reduce these transactions to multiple smaller ones, each with only one coherence event, introducing new intermediate states where necessary.

In principle, there is an intermediate state for every interleaving of in-flight messages and cache states at both nodes—allowing CCKit to handle out-of-order responses and avoid serializing memory transactions. In many cases groups of these theoretical states are indistinguishable at the DC and can be collapsed together, reducing the state space to be explored.

In other cases, to avoid the state machine growing without bounds during state space exploration, it may be necessary for the machine to stall coherence events for a line until outstanding transactions on the line have completed—equivalent to reordering the transaction into a previously known one. Stalling transaction *responses* in this way can lead to deadlocks in the state machine, and so CCKit will only stall requests. CCKit's state machine on Enzian has 79 states with 304 transitions between them and handles 25 different coherence messages.

4.3 Achieving Full Performance

The ThunderX-1 cache controller is heavily pipelined and closely integrated with the cache itself, running at the core clock frequency of 2 GHz. This is more than 6 times the typical FPGA clock of 322 MHz, requiring a tightly optimized design on the FPGA side.

CCKit heavily exploits spatial parallelism to achieve the same throughput at lower clock speed, adopting a design with many simple, non-pipelined units operating in parallel. An example CCKit DC configuration uses 64 in-order Directory Controller Units (DCUs) each divided into odd and even Directory Controller Slices (DCSs) to handle 24 GiB/s of traffic (assuming an AXI-side latency of 300 ns). This is comfortably more than the 70 ns DRAM latency (Figure 5), leaving plenty of overhead for memory-side application logic (e.g., materialized view in Section 5.6).

Each DCU handles coherence for a disjoint subset of cache lines. Physical addresses are thus used to route messages to a DCU. Depending on its type, a coherence message consists of one 64-bit header and up to 16 64-bit payload words. The header contains both the data needed to route to a DCU (address), and all information (message type, dirty bits, etc.) relevant to the state machine.

As shown in Figure 3, header and payload are separated on ingress. The variable-length payload is stored in a BRAM (embedded SRAM blocks) table indexed by DCU number and retrieved only when needed to generate a write transaction on the user-facing AXI interface. Only the fixed-size

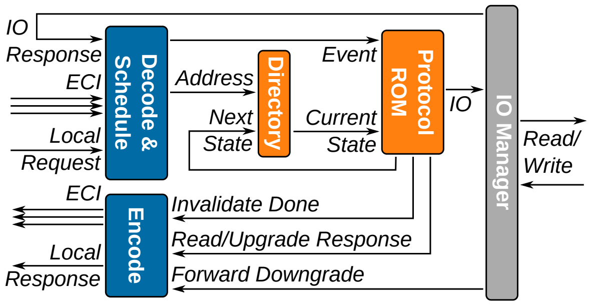

Fig. 4. The Directory Controller Unit (DCU) internals show the read/write interface from the I/O manager, the Protocol ROM, directory, and the decode/encode blocks.

header passes through the routing and arbitration logic. The same separation occurs on the AXI response path (for reads) with the generated ECI header attached to the payload only on egress. AXI transactions are tagged with DCU number to ensure correct routing of responses.

Each DCU is serial and blocking: there is only one outstanding transaction of each read, write and coherence transaction types, however other actions can be taken while the controller itself is blocked. Its serial nature ensures faithfulness of the implementation to its protocol state machine: all steps for an event is completed before the next event is chosen. To avoid resource deadlocks, the DCU never stalls waiting for a resource. Whenever an event cannot be handled, it gets delayed and the DCU tries to handle a different event. Responses are prioritized over requests as they may free up resources, but scheduling is otherwise round-robin. All achieved parallelism is therefore from concurrent transactions on separate DCUs. This works for most workloads as long as the number of DCUs per slice is chosen to avoid blocking for a sequential workload accessing each DCU in turn: other workloads benefit from an XOR-based address scrambler in the CPU's LLC designed to break up pathological address patterns.

Figure 4 shows the internal architecture of a DCU, with the encoding/decoding interface to raw ECI on the left, and the common AXI interface to user logic on the right. The DCU does not generate multi-phase AXI transactions directly, but rather sends a single descriptor per operation and waits on the result. These descriptors pass through an arbitration and routing stage analogous to that for ECI messages. At the AXI port the DCU ID is translated to an AXI ID, allowing in-flight transactions equal to the number of DCUs, which may complete out of order. The DCUs neither assume nor guarantee any ordering on requests to different cache lines.

The directory and protocol ROM (generated from the high-level formal specification in Section 4.2) together implement the per-line coherence protocol. The ROM is indexed by the current state of a line (read from the directory) and the requested operation, and contains the new line state plus any action (e.g., AXI read) required. The directory is also scaled to give good utilization of FPGA resources, given the size of the CPU LLC whose state it tracks, with the same associativity as the LLC to minimize unnecessary invalidation traffic.

4.4 Resource Usage and Footprint

Table 1 shows resources used by different components and configurations of CCKit with 64 DCUs. The first two lines show the individual resource consumption of the ECI transport layer and the DC. The remaining lines show usage when the DC is configured to access BRAM memory or off-chip DDR controllers (via Xilinx-provided "MIGs"). Even with this full configuration, CCKit leaves 70% of the FPGA resources for applications. While the physical limitations of the FPGA does not allow

Table 1. Resource Utilization Footprint on FPGA

Configuration	LUT (%)	CLB (%)	BRAM (%)
ECI	7.90	11.27	8.24
DC	6.86	12.16	5.93
ECI+DC+BRAMs	14.89	24.03	15.65
ECI+DC+MIGs	19.23	30.26	17.33

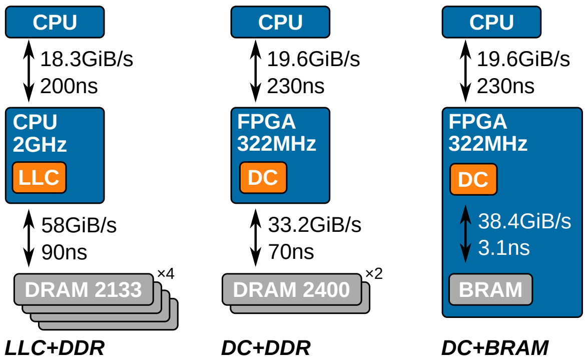

Fig. 5. We compare the bandwidth and latency of the 2x CPU system (LLC+DDR), the CPU-FPGA system with DC and DDR, and the CPU-FPGA system with DC and BRAM.

for unlimited scaling, the newer FPGAs have grown significantly (the VU19P has nearly 9 M CLBs versus the VU9P's 2.5 M). Additionally, there is a trend to greater hardening of critical infrastructure (e.g., memory controllers) to provide better performing and space efficient implementation. Due to these trends, we expect an overall reduction in both relative and absolute usage of resources in the implementation of CCKit.

5 Evaluation and Example Applications

We present both micro-benchmarks and example applications to demonstrate that CCKit provides comparable performance to the native CPU hardware, and that its flexibility and customization enables innovative features beyond simple acceleration. The four different use-cases show CCKit can be used to explore acceleration models enabled by both asymmetric and symmetric coherence. We focus on simplified examples to highlight acceleration patterns enabled by CCKit, rather than accelerating a complete application.

5.1 Experimental Setup

We microbenchmark CCKit on Enzian in two different configurations, shown in Figure 5. In the *DC+DDR* configuration, the FPGA connects the AXI interface of each DCS to one 16 GiB DIMM with a standard Xilinx DRAM controller IP (MIG) [106]. In *DC+BRAM*, we replace the DRAM with two 64 KiB BRAMs to isolate the performance of the DC from that of the Xilinx DRAM IP.

As a baseline, we compare these with *LLC+DDR*, a 2-socket Gigabyte R150-T61 [44] server based on two Marvell CN8890 ThunderX-1 CPUs connected by CCPI, the proprietary native coherence protocol on which ECI is based. As with Enzian, the CPUs run at 2 GHz and each node has four 64 GiB DIMMS.

The round-trip latency and throughput figures in Figure 5 are the measured performance of the existing hardware (for *LLC+DDR*), or the ECI implementation supplied with Enzian (for *DC+DDR*

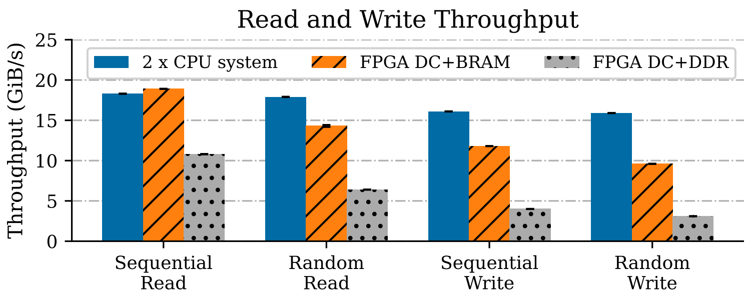

Fig. 6. The CCKit DC with BRAM outperforms the 2x CPU system for sequential reads, while showing decreased performance for other operations. The DDR case performs worse, in part due to latency, but especially from the performance of the MIGs in random accesses.

and *DC+BRAM*). The DRAM and BRAM figures are likewise measured on the unmodified base platform. These thus represent upper bounds on the performance of CCKit's DC as fixed parameters of the underlying platform. ECI shows 7% higher throughput at 15% higher latency than the native 2-socket ThunderX-1 implementation.

5.2 DC Read-Write Performance

While the supplied Enzian ECI implementation is comparable to the CPU's own, the DC adds symmetric coherence between the two sockets. In this section, we evaluate the performance after introducing the DC. For all three configurations in Figure 5, we measure throughput and latency for sequential and random reads and writes on a contiguous 1 GiB region. As the ThunderX-1 LLC has no hardware prefetcher, both sequential and random read throughput tests use prefetch hint instructions to avoid serializing on LLC refills.

Figure 6 presents throughput for all combinations. Each bar is the mean of 100 runs, with (negligible) standard deviation indicated. For *DC+BRAM*, sequential reads slightly exceed the baseline, showing that the distributed DC is able to match the throughput of the CPU's LLC at 1/6 the clock rate. Throughput drops significantly once BRAM is replaced with DRAM. One potential cause is the known inefficiency of the Xilinx MIG IP under non-sequential access patterns [106], interacting with the ThunderX-1 LLC address scrambler, transforming sequential reads into a pseudo-random pattern. Fully random reads further stress the MIG's scheduler and begin to cause contention on DCUs, leading to a moderate slowdown relative to sequential. Future improvement in the Xilinx IP could reduce or eliminate this inefficiency.

Sequential and random writes perform similarly, and are broadly consistent with random reads. This is likely due to ECI writeback messages to the DC being generated not in program order but by LLC evictions which introduce additional randomness to the access order.

The overall trend is lower throughput as randomness increases, consistent with reduced utilization of DCUs, compounded by the low non-sequential performance of the MIG IP. The bursty nature of write traffic from the ThunderX-1's 3 KiB per-core write buffer likely also contributes to exceeding the in-flight transaction capacity of the DC. Increasing the number of outstanding transactions per DCU would improve performance for applications with more random or write-heavy access patterns.

From a design perspective, non-sequential accesses can suffer because each DCU is non-pipelined to optimize for resources. When a DCU is processing a coherence event, subsequent coherence events are blocked until the operation completes. Therefore, best performance is achieved for a sequential workload where all DCUs are busy. In a bursty workload, some DCUs might be idle while

Table 2. Sequential and Random Read Latency

Configuration	Seq. Read (ns)	Rand. Read (ns)
LLC+DDR	268	271
DC+BRAM	454	444
DC+DDR	591	601

Table 3. Directory Controller Clean-Invalidate Performance

Configuration	Throughput (10^6 CL/s)	Latency (ns)
Lower bound	150	–
DC+BRAM	181	350
DC+DDR	230	350
Upper bound	322	–

others might have multiple outstanding messages to be handled. A sequential write workload is a sequential read exclusive into the CPU's cache followed by a bursty write as we do not have control over the order in which cache lines get evicted. Thus even with a BRAM, a drop in performance is observed.

Table 2 shows the average round-trip latency of reads for all three configurations. These are the average of 15 runs over the full 1 GiB, with one access per cache line and no prefetching to ensure serialization. Comparing *LLC+DDR* with *DC+DDR* indicates that CCKit adds 323–330 ns latency relative to the CPU, with the random pattern 1–2% slower. Comparing *DC+BRAM* with *DC+DDR* isolates the impact of the DC itself to 173–186 ns, with the remaining 137–157 ns due to the DRAM latency and the AXI interconnect. A write instruction commits once it hits the write buffer, ergo it has the same latency characteristics as a read.

While DC performance can be further optimized, it already demonstrates that with careful design, CPU-comparable performance can be achieved on FPGAs.

5.3 DC Clean-Invalidate Performance

In addition to allowing the CPU to coherently cache FPGA-homed data, CCKit provides a request-acknowledge interface so that FPGA applications can issue *clean* or *clean-invalidate* requests for FPGA-homed cache lines, and wait until the operation completes. Both operations cause the writeback of an FPGA-homed cache line that is dirty in the CPU's LLC, with clean-invalidate additionally invalidating the CPU's copy. Many use cases described in this article rely on these operations and so we evaluate their performance.

For both *DC+BRAM* and *DC+DDR* configurations, the CPU first reads 8 MiB of sequential data into its LLC in shared (clean) state, which is then invalidated by an FPGA application via the DC. To measure the latency of a single round-trip invalidation, the application issues one outstanding request to the DC at a time. To measure throughput, the application issues as many requests at a time as possible and measures the time taken to invalidate 8 MiB of CPU-cached data. This throughput is higher than indicated by per-request latency due to benefits from pipelining.

Throughput and latency are shown in Table 3. The latency of a single invalidate is the same in both cases (350 ns) but throughput varies. The throughput is bounded above by the rate at which the application can issue requests (1 per clock at 322 MHz), and below by the time needed to write back 8 MiB of dirty 128 B cache lines at the measured ECI throughput of 20 GiB/s ($\approx 150 \times 10^6$ CL/s). The computed throughput varies, as an unpredictable fraction of the lines are voluntarily evicted by the CPU, in which case the DC completes immediately without sending a message, but are in both cases solidly between the indicated bounds. We conclude that for any application with a non-trivial fraction of dirty data in the CPU LLC, the FPGA-initiated invalidation rate will be limited by the bandwidth available for dirty data writeback, and not the DC.

5.4 Concurrent Shared Data Structures Access

As a first application use-case, we demonstrate how to use CCKit to enable threads on the CPU (host) and the FPGA (device) to concurrently work on a shared data structure while maintaining coherence. This application is used to demonstrate CCKit's equilvalent of CXL.cache protocol or the fine-grained acceleration pattern enabled by HARPv2 [19, 27, 29] but with the following distinctions. First, for simultaneous host and device access to device memory, CXL 2.0's assymmetric protocol requires an expensive round-trip interaction over the interconnect for every single cache line access by the device to its own memory. Furthermore, CXL introduces a special non-coherent *bypass* mode for mitigating this cost when the device memory is marked as non-accessible by the host. In contrast, the symmetric nature of ECI initiates traffic over the interconnect *only* when the cache line is cached by the host, removing the need for a *bypass* mode irrespective of whether the device memory is marked as shareable or not.

This example serves as the basis for distributed computing applications like *appbt* [17] and for approximate computing [56] where the data is modified through the network directly on the FPGA while threads on the CPU are also accessing the same data. The experiment also allows us to observe CCKit under contention and demonstrate that it has the same performance characteristics as a conventional NUMA system with two CPUs compared to e.g. HARP in which fine-grained co-processing is considered nearly impossible [27].

The shared data structure we use in this experiment is a table homed in FPGA BRAM. Each row is padded to the cache line size of 128 B. The table has a size of 8 MiB (65536 rows). The CPU and FPGA concurrently scan the table and increment the value of a counter at each row. The CPU always scans the full table, but we vary the contention rate by limiting the FPGA to only access a part of the table. We run the experiment for 1 s and measure the number of rows the CPU is able to process. For comparison we run the same workload on our 2-socket Gigabyte server. The memory for the shared table and the thread generating contention are pinned to one of the NUMA nodes using Linux's NUMA policy library. The thread that always scans the full table is the same as in the CPU-FPGA case and is pinned to the other NUMA node. In both cases we warm the L2 cache on the CPU where we perform the measurement.

The FPGA thread uses the locking variant of clean-invalidate. Once it is completed, cache line state in the CPU's LLC is invalid and the BRAM has the most up-to-date value for this cache line. The FPGA thread can atomically read-modify-write this cache line before unlocking it.

Figure 7 shows the results. We plot the throughput of the thread that always scans the full table vs. the fraction of the table that is accessed by the contending thread. Throughput is given in **millions of rows per second** (**Mrps**), and we report average and standard deviation for 10 iterations. In the CPU-FPGA configuration the CPU reaches about 45.5 Mrps without contention. This gradually degrades to about 2.5 Mrps when the FPGA contends for the entire table. In the CPU-CPU configuration the respective numbers for the access thread are about 41 Mrps with no contention and 5 Mrps with maximum contention.

We can see from the shape of the curves in Figure 7 that the CPU-FPGA setup using CCKit behaves very similarly to a two socket server with the same CPU. The slightly lower performance of the CPU-FPGA configuration is explained by the latency that CCKit adds to migrating a cache line across the interconnect.

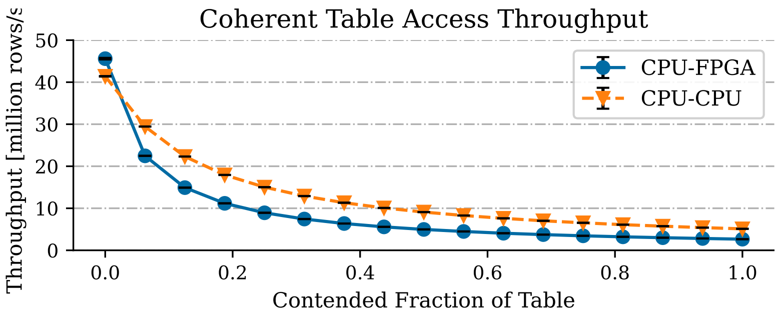

Fig. 7. Sharing a coherent table across two CPUs and the CPU and FPGA shows similar performance, with the FPGA slightly outperforming the CPU-CPU system at very low contention, and nearly matching its performance as table contention increases.

5.5 Application-Specific Prefetching

The second application showcases CCKit's equivalent of CXL.mem protocol for memory expansion. In this use case, we use the FPGA as a smart memory controller that expands the memory available to the host by providing coherent access to device memory while adding new features. This is the basis for application-customized memory controllers that provide support for special data types, memory traversal functions, or for exploring memory-semantic SSDs [110] (prefetch data from storage to FPGA main memory) and near-storage FPGA acceleration [48, 61]. We implement a sequential parallel prefetcher which increases the performance of any accesses to the FPGA-homed in-memory data when multiple concurrent client threads read it sequentially. The use case also illustrates the benefits of tailoring the memory subsystem behavior to specific applications, their access patterns, and how this can be tightly integrated with the coherence stack.

Using the *DC+DDR* configuration, the prefetcher lies between the DC and FPGA-side DRAM, intercepting all reads to tables that miss in the CPU LLC. To maximize performance, we implement multiple parallel prefetch units, each of which can read data for different execution threads using on-chip BRAM to store blocks.

When an LLC miss occurs within the table, the prefetcher first calculates the block address of the intercepted cache line. It then assigns this request to an available prefetch unit which fetches the block from DRAM in a series of 64-beat AXI burst transactions to maximize the available DRAM performance. To avoid unnecessary overheads, prefetching is done in the background, and the intercepted cache line is served and returned to the CPU with priority (i.e., hot line first) through the existing direct path to DRAM. Additionally, when a sequential read to the second half of a block is detected, another prefetch unit is preemptively allocated to get the next block. This overlapping ensures that subsequent access to increasing offsets is served with minimal delay. A non-sequential read from a block that is not currently preloaded will be assigned to a free unit to start a new prefetch operation.

Our experiment reads table rows sequentially from the FPGA's DRAM. The prefetcher we configure has eight parallel blocks of 8 KiB each, which in total uses $< 1\%$ of the available BRAM resources. Overlapped operation is sufficient to sustain effective prefetching for the workloads of up to four concurrent clients. Figure 8 plots throughput against the number of concurrent threads performing the read operation, compared to the performance without the prefetcher. We see that the addition of the prefetcher significantly improves the sequential memory access performance, nearing the DRAM's optimal throughput.

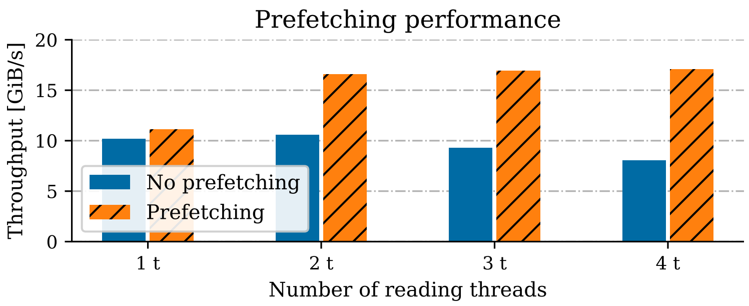

Fig. 8. The application-specific prefetcher demonstrates near-memory acceleration by improving the throughput of a multithreaded application by nearly 2× with 4 threads.

5.6 Materialized Database View Maintenance

In the third use case, we demonstrate how CCKit enables non-traditional FPGA acceleration models that go beyond the traditional models provided by PCI Express (PCIe) and CXL interconnects. We offload application specific coherence onto the FPGA transparently to software running on the CPU. We also use this application to introduce the notion of application-specific coherence protocols and how they can be built on top of standard coherence protocol layers in CCKit.

Such an application could potentially be implemented with CXL 3.0's *back-invalidation snooping* mechanism but since CXL was not designed for customizing coherence protocols (for e.g. the device's home agent still resides across the interconnect on the CPU), it is unclear whether it is possible. Nevertheless, recent works on near-data-processing [62], crash consistency [12], and disaggregated memory [22] propose such an interaction.

We offload view maintenance as used in a relational database to the FPGA and use coherence to ensure the CPU always sees consistent data even as the base table is being modified. Relational engines use views to provide logical data independence: the ability to provide different data organizations over a common underlying schema. Views can be virtual or materialized, meaning that the view corresponds to an actual table that is the result of running the query defined in the view. Such materialized views are used for a range of purposes: access control, simplifying query development, and performance optimization by pre-computing parts of common queries. Here, we exploit the FPGA's ability to control the CPU's cache in an application-specific manner, using the fact that CCKit provides access to coherence protocol messages to trigger operations on the FPGA that handle expensive view maintenance tasks the CPU would otherwise have to perform.

For the experiment, we use a table from the TPC-H benchmark, `ORDERS`, containing information about orders placed by clients. This base table is append-only, and resides in the DRAM of the FPGA. The attributes of interest are `O_CUSTKEY` (customer identifier) and `O_TOTALPRICE` (sale price), both stored as 64 b integers. The table is coherently accessible from the CPU as normal, writable NUMA memory.

We define a materialized view over the base table as follows: `SELECT SUM(O_TOTALPRICE) FROM ORDERS GROUP BY O_CUSTKEY ORDER BY O_CUSTKEY;`. This view aggregates the total price of all orders by each customer, sorting the result by customer key. The materialized view is stored in a second coherent address range backed by FPGA BRAM.

The offloaded view maintenance (i.e., application protocol) works as follows. On the CPU side, transactions update the base table with new orders. Each appends a tuple to the table by loading the next tuple location in exclusive mode into the CPU cache and updating it. Upon transaction commit, a view maintenance operator on the FPGA is triggered. This operator invalidates the CPU cache

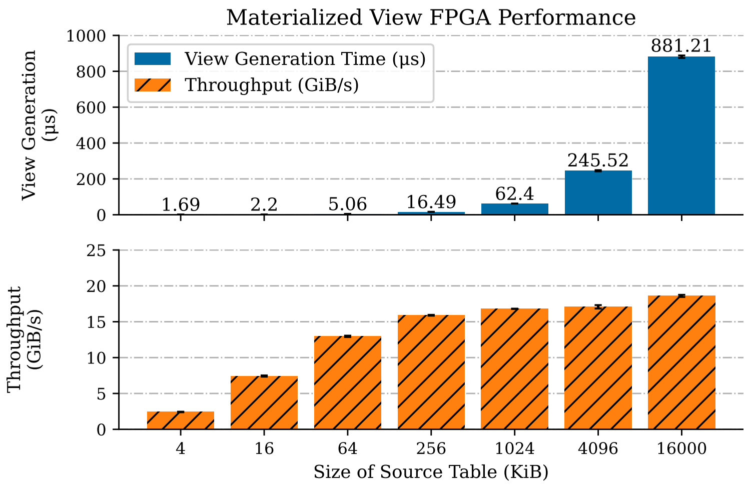

Fig. 9. The performance of the materialized view experiment shows that the materialization operator is bound by interconnect bandwidth. Latency remains under 1ms for up to 16000 B source table.

lines holding updated tuples and, as part of the process, reads the data written back and updates both the base table and the materialized view with the new aggregate calculations. From this point on the view table is consistent with the base table and can be read freely. The CPU invokes the operator by issuing a read on a pre-defined synchronization address, signaling via an invalidation message to run the FPGA's view maintenance operator. Note that the CPU software is no longer required to do any cache or synchronization operations (expensive flushes and fences) to keep two different address spaces (the base and view tables) coherent with each other. The view-maintenance application protocol built on top of CCKit's DC does this transparently.

Figure 9 (*view generation time*) shows how long it takes the FPGA to update the base table and propagate changes to the materialized view. We vary the number of updates (appends) per transaction on the base table and measure the overall throughput observed over the interconnect. As the figure shows, the materialization operator is bound by the interconnect bandwidth, with a response time linear in the base table size since the view is recomputed by recalculating all aggregations. This could be optimized by computing only those that need to be modified and updating the corresponding entries in the materialized view.

This use case shows that CCKit enables the implementation of coherent applications that go beyond the usual definition of coherence, e.g. tying the coherence of multiple addresses related by a computed function.

5.7 Low-Latency Messaging

Lastly, we show that CCKit can be used to implement synchronous, low-latency message passing between coherent agents. Here we build a simple blocking RPC interface between a CPU core and an accelerator on the FPGA. This permits the user to, for example, explore applications enabled by fully-symmetric protocols such as CXL 3.0.

The quiescent state of the protocol, after a registration/initialization phase, involves two cache lines. Both caller (CPU) and callee (FPGA) hold one of these lines in exclusive state, ensuring that a copy exists only in its own cache, and that it may be modified without notification (a silent upgrade). We label the CPU-held line A and the FPGA-held line B.

The CPU begins by writing the RPC arguments to A, which completes locally without any coherence messages. To signal to the FPGA that a request is ready, the CPU issues a load-exclusive

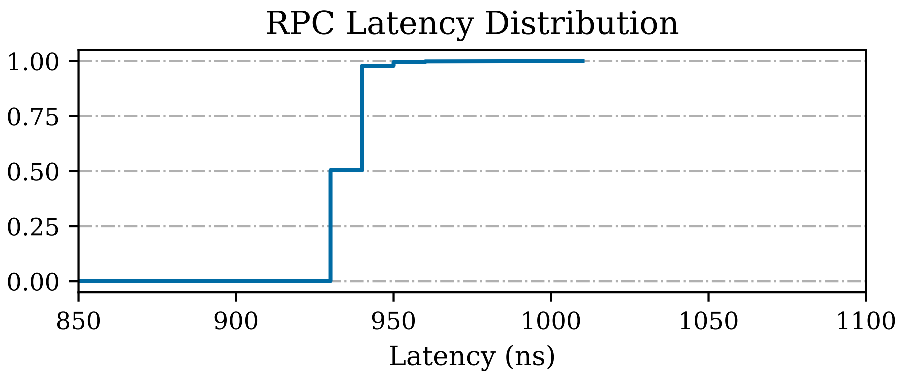

Fig. 10. The latency distribution of an RPC between the CPU and FPGA using CCKit has a median latency of 930ns.

request for B from software, by writing to a reserved field within it arranged to not overlap with the eventual RPC response. This generates the corresponding interconnect message, which the FPGA receives. The CPU core will stall until this message receives a response.

Knowing that A now holds RPC arguments, the FPGA issues its own load-exclusive for A. The CPU's cache responds by forwarding the line's contents to the FPGA and invalidating its own copy. The FPGA now holds both lines exclusively, and has the RPC arguments. The FPGA does not need to implement a full cache, and simply uses a pair of registers to hold the content of both lines.

The CPU will remain stalled, and the caller thus blocked, until the FPGA responds. In this way the user-specified RPC handler has full control over the protocol's progress. The only limitation is any timeout mechanism on the CPU, which on the ThunderX-1 is on the order of a second. The FPGA need only ensure it completes any processing within this timeout, and everything will remain entirely transparent to software on the CPU.

Once the RPC is completed and its return value available, the FPGA writes this to B and replies to the CPU's stalled load-exclusive request. The RPC result thus ends in the CPU's cache, with the caller unblocked and ready to proceed.

The whole process required two interconnect roundtrips: the CPU's (stalled) request for B, during which the FPGA fetches A. Furthermore, the system is now back to its quiescent state, with each side holding one of the lines in exclusive. The CPU now holds B, and the FPGA A. The next request is thus handled with the two lines exchanging roles.

Figure 10 shows the CDF of end-to-end latencies we measure for a null RPC (the FPGA responds immediately). The latency is extremely predictable and clustered tightly around the median of 930 ns. This corresponds to two interconnect roundtrips. The observed outliers are due to misses in the L1 cache (the protocol itself operates on the L2). Additional disruptions can occur due to the Linux scheduler.

This example protocol is fully serializing and thus limited by the roundtrip time to a bandwidth of around 200 MiB/s. If bandwidth, rather than latency, is the bottleneck for a particular application, the protocol is straightforwardly extensible to transfer as many cache lines as desired in each exchange. Synchronization still occurs over a single pair of control lines (A and B), with payload lines sent first and a barrier ensuring that reordering does not affect correctness. A payload of 100 lines (or around 13 KiB) is sufficient to saturate the Enzian interconnect. These payload lines can also be used to batch requests.

As systems become increasingly power-limited, the cost of dedicating cores to specific tasks (thus risking under-utilization) is decreasing over time. We have long since passed the point where it is efficient (or even possible) to use all functional units on a processor simultaneously at full performance. For example, using the AVX-512 vector units on an x86 processor generally causes the processor to automatically reduce its clockspeed to avoid exceeding its maximum power rating [47]. It is not uncommon for even HPC workloads to actually perform better using the older less-aggressive instruction sets thanks to the higher available clock speed.

It is moreover a common pattern in high-performance user-space IO stacks such as DPDK or SPDK to dedicate a core to polling descriptor queues. The protocol described here improves on this by keeping the processor pipeline stalled whenever the protocol is idle, giving it the opportunity to enter a sleep state.

Where multiple hardware execution contexts (e.g., hyperthreads) are available per core, a single core would be able to coordinate multiple RPC transactions in parallel without increasing its power consumption. The limitation here is the number of outstanding loads the core permits (i.e., the number and/or width of load-store units). This suggests an interesting direction in architecture research for communication-limited workloads, an insight that has also been explored for graph-processing architectures by Intel's PIUMA design [1] and advocated for systems more generally [51].

CCKit achieves this without any additional hardware, beyond the coherent CPU and accelerator, in contrast to systems such as HyperPlane [72]. The protocol we present here also improves on state-of-the-art CPU-based messaging protocols such as FastForward [43], as low-level access to the protocol permits the exchange in only two interconnect round-trips, in a way which cannot easily be replicated in software.

6 Related Work

As discussed in Section 2, there is growing interest in new coherence models and associated applications. Researchers have demonstrated the need for non-standard or even dynamically customized coherence protocols. For example, Cohmeleon demonstrates that, for different types of accelerators, the best performing cache coherence protocol varies at runtime [116]. Similarly, CoNDA demonstrates the benefits of finer-grained coherence, and proposes a more customizable protocol to increase efficiency and performance [14].

FPGAs have been used to optimize a number of algorithms; many of these could greatly benefit from coherence provided by CCKit. For example, FPGAs have been used to efficiently balance a tree data structure [114]. The addition of cache coherence would allow for concurrent access during rebalancing without the need for external signaling or explicit data transfers. Similarly, many features of Alibaba's OLTP X-Engine [50] could benefit from customizable cache coherence protocols, including operators explored in Section 5.

The movement toward data center disaggregation raises questions on how to handle the additional complexity of new memory tiers. Both POND [66] and TPP [69] are built around CXL, but are primarily interested in the near-NUMA latency of the interconnect and not coherence *per se*. However, others have demonstrated the utility of fine-grained cache coherence in disaggregated systems [22, 63]. For example, MIND advocates a flexible cache coherence protocol integrated into the network [63]. Clio argues that customizable, application-accessible coherence is desirable in these systems for limiting coherence overhead [49].

CCKit can be used to prototype memory-semantic storage systems [110], currently only simulated by researchers. Furthermore, cutting-edge memory semantic SSDs [110] do not have an FPGA on the data path, which can be useful to accelerate near-storage data analytics [61] or offload memory management tasks which are critical for inference in billion-parameter LLM models that do not fit in main memory [4].

SmartNICs often employ FPGAs to accelerate common networking tasks such as RPC calls [60] or RDMA [90, 111]. These systems provide significant improvement, but the addition of coherence using techniques provided by CCKit can provide added benefit. For example, in Dagger, coherence could allow the use of low-latency synchronization primitives instead of complex application-level interactions [60]. StRoM, when ported to CCKit, could enable RDMA atomic operations by directly manipulating the cache using customizable cache coherence. Rambda proposes several architectural changes for accelerating memory-intensive applications which are centered around accelerator coherence [111].

The RPC application of Section 5.7 has a lot in common with the concurrently-developed and published CC-NIC [88] design. The CC-NIC authors have come to many of the same conclusions about the potential benefits of the careful use of coherence traffic for efficient message passing. Both it and our example build on the insights of existing work such as FastForward [43] on fast coherence-based software message passing. In our example we take one step beyond CC-NIC in using pipeline stalls as a hardware blocking mechanism, which is enabled by the extremely fine-grained protocol control permitted by CCKit.

A complete discussion of cache coherence simulators is beyond the scope of this section (see [15] for a more thorough discussion). Simulation tools [13, 53, 87] are essential for developing protocols and architectures. However, simulating a real application with these tools is incredibly slow, and often simulators tradeoff architectural fidelity and accuracy for speed [76]. To evaluate the low-level correctness of controllers and protocols, RTL simulations are often necessary, requiring HDL descriptions of the CPU, interconnect, and accelerator which are seldom available to researchers. Even if these models are available, cutting-edge cycle accurate simulators run in the scale of kHz [37, 99], making the simulation of complex systems and applications under real workloads nearly impossible. Synthesizing RISC-V cores on FPGAs running at ≈50 MHz [57, 109] allows for experimentation with full system software stacks at interactive speeds, however it is limited to only a few out-of-order cores per FPGA.

CCKit complements these techniques by providing a real-world implementation that can faithfully interact with not only real hardware (e.g., off-chip memory, accelerators) and software, but as a part of a networked or rack-scale system.

Similarly, the generation of complex but correct coherence protocols, controllers, and NoCs, as discussed in Section 2.3, is an important and active area of research; a complete discussion is beyond the scope of this article. These systems are focused on the low-level architectural decisions when creating (sometimes heterogeneous) SoCs. Many of the techniques used for generation are complimentary [18], however, the scope is significantly different. CCKit aims to explore enterprise and cloud workloads requiring large server-class CPUs and commercial accelerators/peripherals with the ability to expand to the rack scale.

Finally, FPGA shells [58, 59, 67, 112, 113] provide, to varying degrees, spatial and temporal multiplexing of FPGA resources (including externally-attached memory) between applications implemented in user logic, memory translation, and other services such as networking. All target PCIe-based accelerator cards, adopting a DMA-based approach to acceleration, which rules out both the straightforward use of cache coherence between FPGA and CPU, and the flexibility afforded to applications which have direct access to the coherence protocol. CCKit rectifies this, as a potential component of an FPGA OS which implements cache coherence memory access to both FPGA and CPU memory, and as a critical OS abstraction to make coherence protocols accessible to developers of heterogeneous CPU-FPGA applications. It also exposes limitations of existing operating systems when dealing with modern accelerators (Section 3.4).

7 Conclusions

CCKit shows that experimenting with direct access to a real, native cache coherence protocol from FPGA-based user applications is possible using open hardware available today. This fast, portable hardware interface to such a protocol provides the functionality needed for interesting use-cases beyond simple coherence without exposing the complexity of the underlying protocol. This work has been used to evaluate smartNICs [108] and novel memory systems [103]. The whole of CCKit is publicly available as open source as are all the use-cases and benchmarks in this article.[1] Even if CXL becomes the standard interconnect for accelerators, there remains a long term need for observable and customizable tools for exploring coherence in large-scale systems.

Acknowledgments

The authors would like to thank AMD and Cavium for their financial contributions and technical assistance to the Enzian project. We also wish to acknowledge the efforts of many students over the years who contributed to the success of this work.

References

[1] Sriram Aananthakrishnan, Nesreen K. Ahmed, Vincent Cave, Marcelo Cintra, Yigit Demir, Kristof Du Bois, Stijn Eyerman, Joshua B. Fryman, Ivan Ganev, Wim Heirman, et al. 2020. PIUMA: Programmable integrated unified memory architecture. arXiv:2010.06277 [cs.AR]. Retrieved from https://arxiv.org/abs/2010.06277

[2] Reto Achermann, Nora Hossle, Lukas Humbel, Daniel David Schwyn, David A. Cock, and Timothy Roscoe. 2019. A least-privilege memory protection model for modern hardware. arXiv:1908.08707. Retrieved from http://arxiv.org/abs/1908.08707

[3] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler. 2016. Selective GPU caches to eliminate CPU-GPU HW cache coherence. *IEEE International Symposium on High Performance Computer Architecture (HPCA).* Barcelona, Spain, 494–506. DOI: 10.1109/HPCA.2016.7446089

[4] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C. Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. LLM in a flash: Efficient large language model inference with limited memory. arXiv:2312.11514 [cs.CL]. Retrieved from https://arxiv.org/abs/2312.11514

[5] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. 2018. Spandex: A flexible interface for efficient heterogeneous coherence. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).* 261–274. DOI: https://doi.org/10.1109/ISCA.2018.00031

[6] Amazon Web Services. 2020. Amazon EC2 F1 Instances: Enable faster FPGA accelerator development and deployment in the cloud. Retrieved March 4, 2024 from https://aws.amazon.com/ec2/instance-types/f1/

[7] Jonathan Balkind, Ting-Jung Chang, Paul J. Jackson, Georgios Tziantzioulis, Ang Li, Fei Gao, Alexey Lavrov, Grigory Chirkov, Jinzheng Tu, Mohammad Shahrad, and David Wentzlaff. 2020. OpenPiton at 5: A nexus for open and agile hardware design. *IEEE Micro* 40, 4 (2020), 22–31. DOI: https://doi.org/10.1109/MM.2020.2997706

[8] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, et al. 2020. BYOC: A "bring your own core" framework for heterogeneous-ISA research. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20).* ACM, New York, NY, USA, 699–714. DOI: https://doi.org/10.1145/3373376.3378479

[9] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09).* ACM, New York, NY, USA, 29–44. DOI: https://doi.org/10.1145/1629575.1629579

[10] Noah Beck, Sean White, Milam Paraschou, and Samuel Naffziger. 2018. "Zeppelin": An SoC for multichip architectures. In *Proceedings of the 2018 IEEE International Solid - State Circuits Conference-(ISSCC).* 40–42. DOI: https://doi.org/10.1109/ISSCC.2018.8310173

[11] Berkeley Architecture Research. 2022. TileLink. Retrieved June 7, 2025 from https://bar.eecs.berkeley.edu/projects/tilelink.html

[12] Ankit Bhardwaj, Todd Thornley, Vinita Pawar, Reto Achermann, Gerd Zellweger, and Ryan Stutsman. 2022. Cache-coherent accelerators for persistent memory crash consistency. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems* (Virtual Event) *(HotStorage'22).* ACM, New York, NY, USA, 37–44. DOI: https://doi.org/10.1145/3538643.3539752

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The Gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. DOI: https://doi.org/10.1145/2024716.2024718

[1] https://gitlab.inf.ethz.ch/project-openenzian

[14] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, et al. 2019. CoNDA: Efficient cache coherence support for near-data accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture.* 629–642.

[15] Hadi Brais, Rajshekar Kalayappan, and Preeti Ranjan Panda. 2020. A survey of cache simulators. *ACM Comput. Surv.* 53, 1, Article 19 (Feb. 2020), 32 pages. DOI: https://doi.org/10.1145/3372393

[16] Richard Braun, Abishek Ramdas, Michal Friedman, and Gustavo Alonso. 2023. PLayer: Expanding coherence protocol stack with a persistence layer. In *Proceedings of the 1st Workshop on Disruptive Memory Systems (DIMES'23).* ACM, New York, NY, USA, 8–15. DOI: https://doi.org/10.1145/3609308.3625270

[17] David Brooks and Margaret Martonosi. 1999. Implementing application-specific cache-coherence protocols in configurable hardware. In *Proceedings of the International Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing.* Springer, 181–195.

[18] Anastasiia Butko, Albert Chen, David Donofrio, Farzad Fatollahi-Fard, and John Shalf. 2018. Open2C: Open-source generator for exploration of coherent cache memory subsystems. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'18).* ACM, New York, NY, USA, 311–317. DOI: https://doi.org/10.1145/3240302.3270314

[19] Anthony M. Cabrera and Roger D. Chamberlain. 2019. Exploring portability and performance of OpenCL FPGA kernels on intel HARPv2. In *Proceedings of the International Workshop on OpenCL (IWOCL'19).* ACM, New York, NY, USA, Article 3, 10 pages. DOI: https://doi.org/10.1145/3318170.3318180

[20] Irina Calciu, Jayneel Gandhi, Aasheesh Kolli, and Pratap Subrahmanyam. 2020. Using cache coherent FPGAs to accelerate remote access. US Patent US10761984B2, Filed July 27th., 2018, Issued Sep. 1st., 2020. Retrieved from https://patents.google.com/patent/US10761984B2/en

[21] Irina Calciu, Jayneel Gandhi, Aasheesh Kolli, and Pratap Subrahmanyam. 2020. Using cache coherent FPGAs to track dirty cache lines. Worldwide Patent WO2020023791A1, Filed July 25th., 2018, Published Jan. 30th., 2020. Retrieved from https://patents.google.com/patent/WO2020023791A1

[22] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021).* ACM, New York, NY, USA, 79–92. DOI: https://doi.org/10.1145/3445814.3446713

[23] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. 2019. Project PBerry: FPGA acceleration for remote memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19).* ACM, New York, NY, USA, 127–135. DOI: https://doi.org/10.1145/3317550.3321424

[24] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49).* IEEE, Article 7, 13 pages.

[25] CCIX Consortium and others. 2019. Cache Coherent Interconnect for Accelerators (CCIX). Retrieved September 5, 2025 from http://www.ccixconsortium.com

[26] Lucian M. Censier and Paul Feautrier. 1978. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.* C-27, 12 (1978), 1112–1118. DOI: https://doi.org/10.1109/TC.1978.1675013

[27] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2020. Is FPGA useful for hash joins?. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* https://www.cidrdb.org/cidr2020/papers/p27-chen-cidr20.pdf

[28] Jongsok Choi, Ruolong Lian, Zhi Li, Andrew Canis, and Jason Anderson. 2018. Accelerating memcached on AWS cloud FPGAs. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2018).* ACM, New York, NY, USA, Article 2, 8 pages. DOI: https://doi.org/10.1145/3241793.3241795

[29] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *Proceedings of the 2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC).* IEEE, 1–6. DOI: https://doi.org/10.1145/2897937.2897972

[30] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, et al. 2018. Serving DNNs in real time at datacenter scale with project brainwave. In *IEEE Micro* 38, 2 (2018), 8–20. DOI: 10.1109/MM.2018.022071131

[31] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, et al. 2022. Enzian: An open, general CPU/FPGA platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022).* ACM, New York, NY, USA, 590–607. DOI: https://doi.org/10.1145/3503222.3507742

[32] Henry Cook, Wesley Terpstra, and Yunsup Lee. 2017. Diplomatic design patterns: A tilelink case study. In *Proceedings of the 1st Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*.

[33] Intel Corporation. 2009. An introduction to the Intel Quickpath Interconnect. Retrieved September 5, 2025 from https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html

[34] CXL Consortium. 2020. Compute Express Link. Retrieved September 5, 2025 from https://www.computeexpresslink.org/

[35] CXL Consortium. 2020. CXL Webinar: Introduction to Compute Express Link (CXL). Retrieved September 5, 2025 from https://youtu.be/RpAshNmpqLQ?t=1856

[36] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, et al. 2018. The celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro* 38, 2 (2018), 30–41. DOI: https://doi.org/10.1109/MM.2018.022071133

[37] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R. Larus. 2023. Manticore: Hardware-accelerated RTL simulation with static bulk-synchronous parallelism. arXiv:2301.09413. Retrieved from https://arxiv.org/abs/2301.09413

[38] Farzad Fatollahi-Fard, David Donofrio, George Michelogiannakis, and John Shalf. 2016. OpenSoC fabric: On-chip network generator. In *Proceedings of the 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 194–203. DOI: https://doi.org/10.1109/ISPASS.2016.7482094

[39] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, USA, 51–64.

[40] Denis Foley and John Danskin. 2017. Ultra-performance pascal GPU and NVLink interconnect. *IEEE Micro* 37, 2 (2017), 7–17. DOI: https://doi.org/10.1109/MM.2017.37

[41] Zexin Fu, Mingzi Wang, Yihai Zhang, and Zhangxi Tan. 2023. Cache coherent framework for RISC-V many-core systems. In *Proceedings of the 7th Workshop on Computer Architecture Research with RISC-V (CARRV 2023)* (June 2023). Retrieved from https://carrv.github.io/2023/papers/CARRV2023_paper_3_Fu.pdf

[42] Gen-Z Consortium. 2020. Gen-Z Core Specification 1.1. Retrieved from https://computeexpresslink.org/resource/gen-z-specification-archive/. Accessed 2025-09-05.

[43] John Giacomoni, Tipp Moseley, and Manish Vachharajani. 2008. FastForward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*. ACM, New York, NY, USA, 43–52. DOI: https://doi.org/10.1145/1345206.1345215

[44] GIGA-BYTE Technology Co., Ltd. 2023. R150-T61 rev. 110) 2U ARM Rackmount Server. Retrieved from https://www.gigabyte.com/Enterprise/ARM-Server/R150-T61-rev-110

[45] Davide Giri, Paolo Mantovani, and Luca P. Carloni. 2018. Accelerators and coherence: An SoC perspective. *IEEE Micro* 38, 6 (2018), 36–45. DOI: https://doi.org/10.1109/MM.2018.2877288

[46] Davide Giri, Paolo Mantovani, and Luca P. Carloni. 2018. NoC-based support of heterogeneous cache-coherence models for accelerators. In *Proceedings of the 2018 12th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. 1–8.

[47] Mathias Gottschlag, Tim Schmidt, and Frank Bellosa. 2020. AVX overhead profiling: How much does your fast code slow you down?. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'20)*. ACM, New York, NY, USA, 59–66. DOI: https://doi.org/10.1145/3409963.3410488

[48] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access, high-performance memory disaggregation with DirectCXL. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 287–294. Retrieved from https://www.usenix.org/conference/atc22/presentation/gouk

[49] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*. ACM, New York, NY, USA, 417–433. DOI: https://doi.org/10.1145/3503222.3507762

[50] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*.

[51] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2021. A case against (most) context switches. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'21)*. ACM, New York, NY, USA, 17–25. DOI: https://doi.org/10.1145/3458336.3465274

[52] Intel. 2020. Intel Agilex FPGA Product Brief. Retrieved from https://www.intel.com/content/www/us/en/products/docs/programmable/agilex-fpga-product-brief.html
[53] Aamer Jaleel, Robert S. Cohn, Chi-Keung Luk, and Bruce Jacob. 2008. CMP $im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*. 28–36.
[54] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-hardware collaborative memory disaggregation and computation for billion-scale approximate nearest neighbor search. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 585–600. Retrieved from https://www.usenix.org/conference/atc23/presentation/jang
[55] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, USA, 1–12. DOI: https://doi.org/10.1145/3079856.3080246
[56] Henry Kao. 2020. *Cache Coherence for Approximate Computing*. University of Toronto (Canada).
[57] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 29–42. DOI: https://doi.org/10.1109/ISCA.2018.00014
[58] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, USA, 107–127.
[59] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 991–1010. Retrieved fromhttps://www.usenix.org/conference/osdi20/presentation/roscoe
[60] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: Efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM, New York, NY, USA, 36–51. DOI: https://doi.org/10.1145/3445814.3446696
[61] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. 2020. SmartSSD: FPGA accelerated near-storage data analytics on SSD. *IEEE Comput. Archit. Lett.* 19, 2 (2020), 110–113. DOI: https://doi.org/10.1109/LCA.2020.3009347
[62] Sangjin Lee, Alberto Lerner, Philippe Bonnet, and Philippe Cudré-Mauroux. 2024. Database kernels: Seamless integration of database systems and fast storage via CXL. In *Proceedings of the 14th Conference on Innovative Data Systems Research, CIDR*. 9–12.
[63] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. ACM, New York, NY, USA, 488–504. DOI: https://doi.org/10.1145/3477132.3483561
[64] Ang Li, August Ning, and David Wentzlaff. 2023. Duet: Creating harmony between processors and embedded FPGAs. In *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 745–758. DOI: https://doi.org/10.1109/HPCA56546.2023.10070989
[65] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Trans. Parallel Distrib. Syst.* 31, 1 (2020), 94–110. DOI: https://doi.org/10.1109/TPDS.2019.2928289
[66] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. ACM, New York, NY, USA, 574–587. DOI: https://doi.org/10.1145/3575693.3578835
[67] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A hypervisor for shared-memory FPGA platforms. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. ACM, New York, NY, USA, 827–844. DOI: https://doi.org/10.1145/3373376.3378482
[68] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. 2020. Agile SoC development with open ESP. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD'20)*. ACM, New York, NY, USA, Article 96, 9 pages. DOI: https://doi.org/10.1145/3400302.3415753

[69] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. ACM, New York, NY, USA, 742–755. DOI: https://doi.org/10.1145/3582016.3582063

[70] Microsoft Azure. 2023. Microsoft Azure Boost. Retrieved from https://learn.microsoft.com/en-us/azure/azure-boost/overview

[71] Seung Won Min, Sitao Huang, Mohamed El-Hadedy, Jinjun Xiong, Deming Chen, and Wen-mei Hwu. 2019. Analysis and optimization of I/O cache coherency strategies for SoC-FPGA device. In *Proceedings of the 2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 301–306. DOI: https://doi.org/10.1109/FPL.2019.00055

[72] Amirhossein Mirhosseini, Hossein Golestani, and Thomas F. Wenisch. 2020. HyperPlane: A scalable low-latency notification accelerator for software data planes. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 852–867. DOI: https://doi.org/10.1109/MICRO50266.2020.00074

[73] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence* (2nd ed.). Springer Cham. DOI: https://doi.org/10.1007/978-3-031-01764-3

[74] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence* (2nd ed.). Morgan and Claypool.

[75] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue* 6, 2 (Mar. 2008), 40–53. DOI: https://doi.org/10.1145/1365490.1365500

[76] Tony Nowatzki, Jaikrishnan Menon, Chen-Han Ho, and Karthikeyan Sankaralingam. 2015. Architectural simulators considered harmful. *IEEE Micro* 35, 6 (2015), 4–12. DOI: https://doi.org/10.1109/MM.2015.74

[77] Neal Oliver, Rahul R. Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, et al. 2011. A reconfigurable computing system based on a cache-coherent fabric. In *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs (RECONFIG'11)*. IEEE Computer Society, USA, 80–85. DOI: https://doi.org/10.1109/ReConFig.2011.4

[78] Lena E. Olson, Mark D. Hill, and David A. Wood. 2017. Crossing guard: Mediating host-accelerator coherence interactions. *SIGARCH Comput. Archit. News* 45, 1 (Apr. 2017), 163–176. DOI: https://doi.org/10.1145/3093337.3037715

[79] Tarikul Islam Papon, Ju Hyoung Mun, Shahin Roozkhosh, Denis Hoornaert, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. 2023. Relational fabric: Transparent data transformation. In *Proceedings of the 2023 IEEE International Conference on Data Engineering (ICDE)* .

[80] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, et al. 2020. BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs. *IEEE Micro* 40, 4 (2020), 93–102. DOI: https://doi.org/10.1109/MM.2020.2996145

[81] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 13–24. DOI: https://doi.org/10.1145/2678373.2665678

[82] Marjan Radi, Wesley W. Terpstra, Paul Loewenstein, and Dejan Vucinic. 2019. OmniXtend: Direct to caches over commodity fabric. In *Proceedings of the 2019 IEEE Symposium on High-Performance Interconnects (HOTI)*. 59–62. DOI: https://doi.org/10.1109/HOTI.2019.00027

[83] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, et al. 2021. Warehouse-scale video acceleration: Co-design and deployment in the wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 600–615. DOI: https://doi.org/10.1145/3445814.3446723

[84] C. Ravishanicar and James R. Goodman. 1983. Cache implementation for multiple microprocessors. In *Proceedings of the 26th IEEE Computer Society International Conference (COMCON)*.

[85] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. 1996. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*. ACM, New York, NY, USA, 34–43. DOI: https://doi.org/10.1145/232973.232979

[86] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. 2020. HMG: Extending cache coherence protocols across modern hierarchical multi-GPU systems. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 582–595. DOI: https://doi.org/10.1109/HPCA47549.2020.00054

[87] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, USA, 475–486. DOI: https://doi.org/10.1145/2485922.2485963

[88] Henry N. Schuh, Arvind Krishnamurthy, David Culler, Henry M. Levy, Luigi Rizzo, Samira Khan, and Brent E. Stephens. 2024. CC-NIC: A cache-coherent interface to the NIC. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS'24)*. ACM, New York, NY, USA, 52–68. DOI: https://doi.org/10.1145/3617232.3624868

[89] Debendra Das Sharma and Ishwar Agarwal. 2022. *Compute Express Link 3.0 Standard*. Technical Report. Compute Express Link Consortium Inc.

[90] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart remote memory. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*. ACM, New York, NY, USA, Article 29, 16 pages. DOI: https://doi.org/10.1145/3342195.3387519

[91] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.* 12, 3 (2010), 66.

[92] Jeffrey Stuecheli, Bart Blaner, C. R. Johns, and M. S. Siegel. 2015. CAPI: A coherent accelerator processor interface. *IBM J. Res. Dev.* 59, 1 (2015), 7:1–7:7. DOI: https://doi.org/10.1147/JRD.2014.2380198

[93] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison. 2018. IBM POWER9 Opens up a new era of acceleration enablement: OpenCAPI. *IBM J. Res. Dev.* 62, 4–5 (July 2018), 8:1–8:8. DOI: https://doi.org/10.1147/JRD.2018.2856978

[94] Sajjad Tamimi, Florian Stock, Andreas Koch, Arthur Bernhardt, and Ilia Petrov. 2022. An evaluation of using CCIX for cache-coherent host-FPGA interfacing. In *Proceedings of the 2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[95] C. K. Tang. 1976. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition (AFIPS'76)*. Association for Computing Machinery, New York, NY, USA, 749–753. https://doi.org/10.1145/1499799.1499901

[96] Wesley W. Terpstra. 2017. TileLink: A free and open-source, high-performance scalable cache-coherent fabric designed for RISC-V. In *Proceedings of the 7th RISC-V Workshop*.

[97] The Enzian Project. 2024. Enzian. Retrieved September 5, 2025 from https://enzian.systems/

[98] Neil C. Thompson and Svenja Spanuth. 2021. The decline of computers as a general purpose technology. *Commun. ACM* 64, 3 (Feb. 2021), 64–72. DOI: https://doi.org/10.1145/3430936

[99] Verilator. 2024. Verilator RTL Simulator. Retrieved March 7, 2025 from https://veripool.org/verilator/documentation/

[100] Tianrui Wei, Nazerke Turtayeva, Marcelo Orenes-Vera, Omkar Lonkar, and Jonathan Balkind. 2023. Cohort: Software-oriented acceleration for heterogeneous SoCs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. ACM, New York, NY, USA, 105–117. DOI: https://doi.org/10.1145/3582016.3582059

[101] Ying Wei, Yi Chieh Huang, Haiming Tang, Nithya Sankaran, Ish Chadha, Dai Dai, Olakanmi Oluwole, Vishnu Balan, and Edward Lee. 2023. 9.3 NVLink-C2C: A coherent off package chip-to-chip interconnect with 40gbps/pin single-ended signaling. In *Proceedings of the 2023 IEEE International Solid-State Circuits Conference (ISSCC)*. 160–162. DOI: https://doi.org/10.1109/ISSCC42615.2023.10067395

[102] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C. Hoe. 2016. A study of pointer-chasing performance on shared-memory processor-FPGA systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16)*.

[103] Xiaoxiang Wu, Baptiste Lepers, and Willy Zwaenepoel. 2025. Pre-stores: Proactive software-guided movement of data down the memory hierarchy. In *Proceedings of the 20th European Conference on Computer Systems (EuroSys'25)*. ACM, New York, NY, USA, 1161–1176. DOI: https://doi.org/10.1145/3689031.3696097

[104] Mark Wyse, Daniel Petrisko, Farzam Gilani, Yuan-Mao Chueh, Paul Gao, Dai Cheol Jung, Sripathi Muralitharan, Shashank Vijaya Ranga, Mark Oskin, and Michael Taylor. 2022. The BlackParrot BedRock cache coherence system. arXiv:2211.06390 [cs.AR]. Retrieved from https://arxiv.org/abs/2211.06390

[105] Xilinx. 2021. UltraScale Architecture and Product Data Sheet: Overview. Retrieved May 24, 2024 from https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[106] Xilinx. 2022. *UltraScale Architecture-Based FPGAs Memory IP (PG150)*. Technical Report. https://docs.amd.com/r/en-US/pg150-ultrascale-memory-ip

[107] Xilinx. 2022. Xilinx Zynq Ultrascale+ MPSoC. Retrieved September 5, 2025 from https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[108] Pengcheng Xu and Timothy Roscoe. 2025. The NIC should be part of the OS. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems (HotOS'25)*. ACM, New York, NY, USA, 151–157. DOI: https://doi.org/10.1145/3713082.3730388

[109] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, et al. 2022. Towards developing high performance RISC-V processors using agile methodology. In *Proceedings of the 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1178–1199. DOI: https://doi.org/10.1109/MICRO56248.2022.00080

[110] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. 2023. Overcoming the memory wall with CXL-enabled SSDs. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC 23).* USENIX Association, Boston, MA, 601–617. Retrieved fromhttps://www.usenix.org/conference/atc23/presentation/yang-shao-peng

[111] Yifan Yuan, Jinghan Huang, Yan Sun, Tianchen Wang, Jacob Nelson, Dan R. K. Ports, Yipeng Wang, Ren Wang, Charlie Tai, and Nam Sung Kim. 2023. Rambda: RDMA-driven acceleration framework for memory-intensive μs-scale datacenter applications. In *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* 499–515. DOI: https://doi.org/10.1109/HPCA56546.2023.10071127

[112] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the cloud. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20).* ACM, New York, NY, USA, 845–858. DOI: https://doi.org/10.1145/3373376.3378491

[113] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. 2017. The feniks FPGA operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys'17).* ACM, New York, NY, USA, Article 22, 7 pages. DOI: https://doi.org/10.1145/3124680.3124743

[114] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. FPGA-accelerated compactions for LSM-based key-value store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies, FAST 2020.* https://www.usenix.org/conference/fast20/presentation/zhang-teng

[115] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. 2022. Understanding data storage and ingestion for large-scale deep recommendation model training. In *ISCA 2022 - Proceedings of the 49th Annual International Symposium on Computer Architecture (Proceedings - International Symposium on Computer Architecture).* IEEE, 1042–1057. DOI: https://doi.org/10.1145/3470496.3533044

[116] Joseph Zuckerman, Davide Giri, Jihye Kwon, Paolo Mantovani, and Luca P. Carloni. 2021. Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous SoCs. In *MICRO-54: Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21).* ACM, New York, NY, USA, 350–365. DOI: https://doi.org/10.1145/3466752.3480065

Received 6 August 2024; revised 28 July 2025; accepted 16 August 2025