DISS. ETH NO. 29662

CCKit: FPGA acceleration in symmetric coherent heterogeneous platforms

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

ABISHEK RAMDAS

Master of Science in Computer Engineering, NYU

born on 01.10.1988

citizen of India

accepted on the recommendation of

Prof. Dr. Gustavo Alonso (ETH Zurich), examiner
Prof. Dr. Timothy Roscoe (ETH Zurich), co-examiner
Dr. Alberto Lerner (University of Fribourg), co-examiner
Prof. Dr. Ryan Stutsman (University of Utah), co-examiner

2023

Abstract

FPGA-based accelerators are becoming pervasive in the cloud and data centers due to their architectural flexibility (they are used in many different configurations) and functionality (they can be reconfigured and reprogrammed at runtime). At the same time, standards like CXL raise questions of how a cache coherence protocol should be presented to heterogeneous applications running on, for example, a closely-coupled FPGA-based accelerator. To address these questions, we present CCKit, an open-source toolkit comprising a complete cache coherency stack for FPGAs with software support, and enabling interesting and novel designs beyond simple coherence or, indeed, any requirement for caching on the FPGA. We describe the implementation of CCKit in detail, show through benchmarks that it is highly competitive with hardware-based implementations, but also, critically, that it enables important use-cases for CPU-FPGA coherence beyond those supported by emerging standards.

Zusammenfassung

FPGA-basierte Beschleuniger sind in der Cloud und in Rechenzentren aufgrund ihrer architektonischen Flexibilität (sie werden in vielen verschiedenen Konfigurationen verwendet) und ihrer Funktionalität (sie können zur Laufzeit neu konfiguriert und neu programmiert werden) immer beliebter geworden. Gleichzeitig werfen Standards wie CXL die Frage auf, wie ein Cache-Kohärenzprotokoll heterogenen Anwendungen präsentiert werden sollte, die beispielsweise auf einem eng gekoppelten FPGA-basierten Beschleuniger laufen. Um diese Fragen zu beantworten, stellen wir CCKit vor, ein Open-Source-Toolkit, das einen vollständigen Cache-Kohärenz-Stack für FPGAs mit Softwareunterstützung umfasst und interessante und neuartige Designs ermöglicht, die über einfache Kohärenz oder überhaupt jede Anforderung für Caching auf dem FPGA hinausgehen. Wir beschreiben die Implementierung von CCKit im Detail, zeigen anhand von Benchmarks, dass es im Vergleich zu hardwarebasierten Implementierungen äußerst konkurrenzfähig ist, aber vor allem auch, dass es wichtige Anwendungsfälle für die CPU-FPGA-Kohärenz ermöglicht, die über die von neuen Standards unterstützten hinausgehen.

Acknowledgments

I am deeply indebted to my advisor and chair of my committee Dr. Gustavo Alonso for giving me this opportunity and for his invaluable patience and feedback. I would like to express my deepest gratitude to my defense committee, Prof. Dr. Timothy Roscoe, Dr. Alberto Lerner and Prof. Dr. Ryan Stutsman, for generously providing their expertise and enthusiasm. Special thanks to Dr. David Cock and Adam Turowski for their insights and guidance that enabled a lot of work presented in this thesis.

I am also grateful to Dr. Ghislain Fourny for his kindness and our tea-time discussions. Thanks should also go to my colleagues, Daniel, Michael G., Andrea, Michal F., Michal W., Zhenhao, Dario, Monica, Fabio, Anastasiia, Wenqi, Lukas, Melissa, Nora, Ben, Tom, Roman, Jasmin, and Marko, for making the Systems Group such a wonderful place to work at. Many thanks to the administrators Simonetta, Nadia, Jena and Natasha for organizing amazing retreats and their help in general.

Finally, I would be remiss not mentioning my spouse Alena, son Roman and my parents for their unconditional love and support.

Contents

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, etc. Such a shift in hardware design is in turn driven by performance scaling [CCP+16, FPM+18], parallel machine learning workloads [CFO+18], and specialization [TS21]. 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 PCI Express (PCIe) lacks the necessary features to support increasingly sophisticated and powerful accelerators, new proposals like CCIX [CCI19], GenZ [Gen20], and OpenCAPI [SSI+18] emerged. These have now converged and been merged into CXL [CXL20a] which seems to have become the agreed upon standard for future interconnects. Similar, more specialized developments exist for GPUs (NVIDIA NVLink [FD17, LSC+20], AMD's Infinity Fabric [BWPN18]), and RISC-V processors and embedded systems (TileLink [Ber22]). 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., [LBH+23]).

These emerging interconnects enable innovative architectural designs exploiting coherence, such as disaggregated memory [CIP+21] or crash consistency for persistent memory [BTP+22]. Some even argue that cache coherence protocols should be tailored to the application [MHEH+19, ZGK+21] 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 [NSHW20a]. Implementing a coherent endpoint as part of an application is difficult and time-consuming. 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, real-world practical issues.

To address this, we present CCKit, an open-source, server-grade, modular, reusable, and highly flexible coherence protocol design and implementation. We focus on Field Programmable Gate Arrays (FPGAs) as their reconfigurable nature is ideal both for exploring the design space and meeting the performance requirements of CPU interaction at such a low level. Many of the proposals taking advantage of coherent interconnects are based on FPGAs [CPK+19, CIP+21, BTP+22] and some compare already patenting use-cases based on cache coherent FPGAs [CGKS20a, CGKS20b]. FPGAs are also a standard component in the cloud (Microsoft [PCC+14], Amazon [Ama23], Alibaba [CLL+18]), with novel applications in, e.g., acceleration of database engines [ZWC+20, PMR+23] that would greatly benefit from using a cache coherent FPGA.

Prototyping with CCKit is fast and faithful: its first implementation already runs natively on a real hardware platform [CRS+22], 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, it is also much easier to use, abstracting most of the state machine complexity of the real coherence protocol while still 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.

We show the performance and versatility of CCKit through micro-benchmarks and a range of acceleration use cases. The former demonstrate that CCKit on an FPGA has performance in the same range as 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 throughput when reading the FPGA memory from the CPU versus no pre-fetching, 17.4 GiB/s vs. 7.8 GiB/s for 4 threads); (b) the maintenance of database views with update propagation from base tables to an aggregated view (running at the interconnect speed of 19.5 GiB/s); and (c) synchronous RPC from CPU to FPGA based on the CCKit directory controller that outperforms both programmed I/O and Direct Memory Access (DMA) with a null RPC time of 1.2 μs.

1.1 Background and Motivation

In this section we briefly discuss the hardware trends behind CCKit to pose the two questions we aim to answer with this thesis. First, how to explore the design space of coherent accelerators without the limitations of simulation – essentially, how to *implement* FPGA applications that interact with a CPU over a coherent interconnect? Second, by taking a wider perspective on the integration of accelerators in the overall architecture, what is the appropriate application *interface* to this interconnect from FPGAs?

1.1.1 Symmetric vs. asymmetric protocols

A crucial aspect in cache coherence protocols is who controls the protocol. In this regard, models of cache coherence broadly fall into two categories [NSHW20b].

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 as well as significantly affecting performance: to access areas of its local memory marked as shared with the CPU, an accelerator must make a request to the CPU. The rationale is that, in the common case, data is copied in bulk to the accelerator in advance of computing on it, as with traditional GPUs (see below). In such a model, the underlying idea is that the data in the accelerator is, from the start, a copy and there is no notion of the accelerator actually owning data that the CPU could be caching as well.

In contrast, *symmetric* protocols have home agents on both CPU and accelerator, essentially as in a homogeneous NUMA system. While more complex to implement, they provide seamless integration between the CPU and accelerator as full coherence protocols. Less obviously, they also allow the accelerator to participate in the protocol in more unconventional ways. Rather than just observing transactions on the CPU cache and be notified of these actions by the CPU, the accelerator can actively generate its own notifications and manage its own memory independently. More details can be found in subsection 2.5.1.

Most of the protocols used to date in accelerators are asymmetric. While CCKit can also support them, our main goal is to support symmetric protocols as they are more powerful and open up far more possibilities. Symmetric protocols also seem to be the direction in which current standards are going, making CCKit the first readily available stack that can be used as a means to explore cache coherence in accelerators in general and *symmetric coherence* in particular, as well as a tool to exploit cache coherence at the application level.

1.1.2 The evolution of interconnects

Until recently, accelerators like GPUs used a “host-device” computational model based on PCIe where 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 [NBGS08], OpenCL [SGS10], and modern accelerators such as TPUs [JYP+17] and VCUs [RSC+21]. 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 [OSC+11].

As accelerators have become more powerful (in some cases, many CPUs are needed to feed a single accelerator [ZAB+22]) the lack of PCIe bandwidth has led to better PCIe standards with much higher bandwidths. However, the underlying principals of PCIe have remained unchanged, despite its limitations in terms of protocol and features, and the proliferation and diversity of accelerators emerging recently.

Early attempts at better ways to integrate accelerators into the system were Intel HARPv2 [OSC+11] and IBM CAPI [SBJS15], which coherently connected a server-class CPU and

FPGA. HARPv2 used an *asymmetric* implementation of the symmetric QuickPath Interconnect (QPI) protocol [Cor09] in contrast to other approaches available at the time [WMW+16], 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 application on the FPGA has only limited access and control over the coherency protocol.

More recent developments in more modern interconnects include the Cache Coherent Interconnect for Accelerators (CCIX) [CCI19] which supports a symmetric protocol by extending PCIe, and OpenCAPI [SSI+18] 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 FPGAs attached using CCIX [TSK+22] have emphasized the importance of cache coherence in heterogeneous architectures involving FPGAs. Compute Express Link (CXL) [CXL20a] builds coherence and memory semantics above PCIe and, to date, provides a unified coherent memory space between CPU and accelerators using an asymmetric protocol with coherence bypass for direct access to unshared device memory. These standards were developed in closely-guarded industrial settings, and are evolving rapidly, having largely converged on CXL. At the time of writing, the first CXL 1.1 hardware is becoming available. Symmetric coherence is planned for CXL 3.0 [SA22], and the advent of the specification has already triggered interesting ideas around what it will allow, based on simulations that promise impressive throughput and latency, as well as extensibility beyond one machine [LBH+23, MWD+23]. However, it remains unclear when suitable 3.0 hardware will appear, and CXL retains a somewhat prescriptive position on the use of cache coherence messages.

1.1.3 Coherence in MPSoCs

Alongside these developments in server architecture, processors have started to evolve towards more eclectic and less monolithic designs using *chiplets* and SoC architecture as the basis for more powerful processors.

The most mature and broadly adopted coherent CPU-FPGA systems combine both on a single Multiprocessor System-on-chip (MPSoC), such as Xilinx Zynq UltraScale+ [Xil22b] and Intel Agelix [Int20]. This 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 Last Level Cache (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 rather than servers.

RISC-V's TileLink takes a more general approach [Ber22, Ter17, CTL17], aimed at low-latency connectivity between CPUs, caches, accelerators, memory, and other SoC components. TileLink comprises three protocols (increasing in functionality): Uncached Lightweight (TL-UL), Uncached Heavyweight (TL-UH), and Cached (TL-C), and a number of coherence *policies* which can be subsets of the MOESI protocol (MESI, MEI, MSI, etc.). Multi-socket coherence can be achieved with TileLink-over-Ethernet (TLoE) with OmniXtend [RTLV19]. While TileLink targets the on-chip memory hierarchy [Ber22] and implementations for high-end server 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 [MHEH+19, ZGK+21, GSL+22].

1.1.4 FPGA operating systems

A final motivation for CCKit is recent work on operating systems for FPGAs. Coyote [KRA20], Optimus [MZL+20], AmorphOS [KLP+18], Feniks [ZXX+17], and ViTAL [ZL20] 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. They all target PCIe-based accelerator cards, and so adopt a DMA-based approach to copying the contents of memory to and from the FPGA. This works well for traditional GPU-style application acceleration, but 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 (subsection 1.2.5).

1.2 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, hiding most of the protocol's complexity. It therefore provides a portable interface for application logic to coherently access memory alongside the CPU and also, crucially, to interact with the CPU's LLC. Behind this interface, we show an efficient, deadlock-free, and scalable design providing access to the full address space while maintaining coherence invariants.

A key idea in CCKit is to factor out as much of the complexity of the coherence protocol into scalable re-usable units (part of the CCKit toolkit). The design is, in principle, applicable to a range of hardware platforms, but any implementation will be specific to a particular CPU, coherence protocol, and platform. The first implementation of CCKit is based on the Enzian platform [CRS$^+$22] (section 3.3).

1.2.1 Target platform 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 MOESI-like directory-based write-invalidate cache coherence protocol connects the two. Physical address space is partitioned between the two nodes (i.e. high-order bits of the physical address determine the node where the memory is accessed). CCKit assumes an architecture-specific layer on the FPGA side which exchanges coherence messages with the CPU, guaranteeing delivery but not order, and deadlock freedom.

These assumptions are reasonable: most modern coherent multi-socket systems adopt a directory-based [Tan76, CF78] approach rather than less-scalable, broadcast-based *snooping* [RG83] protocols. In directory-based systems every cache line has a "home" node, i.e. the location for storing both the line data in main memory and the directory data on where the cache line is held and in what states [NSHW20b]. Write-invalidate protocols include the well-known MSI, MESI, MOESI, MOSI, and Intel MESIF, and most modern systems implement a variant of MOESI, supporting cache-to-cache transfers and multiple, consistent-but-dirty copies of data. More details on target platforms and assumptions are discussed in subsection 3.2.1.

In practice, race conditions, message reordering by the interconnect, and concurrency mean that real implementations have many more hidden, intermediate states than the five textbook MOESI states, greatly complicating the protocol [NSHW20b]. More that than 100 states is not unusual in a multi-socket system, and this complexity is even cited as an argument for using asymmetric protocols or no coherency at all in connecting an FPGA and CPU [CXL20b]. CCKit stands as evidence against this, providing a full symmetric protocol implementation that (as we show in subsection 9.5.3) keeps pace with the native CPU implementation.

1.2.2 Coherence protocol specification

Implementing CCKit requires the full coherence protocol to be specified (indeed, we generate the state machine from a formal specification of the protocol), but within these constraints on hardware the interface provided by CCKit to user logic is completely independent of the protocol details.

The coherence protocol on Enzian is the CPU's native coherence protocol which is proprietary and its specification is not available to us. To overcome this, we reverse engineered the coherence protocol from traces and developed a model of the coherence protocol. Using this model, we were able to identify coherence transactions and develop a specification of the CPU's native coherence protocol. We then expanded this specification to also include the interface provided to user-logic. Then we built a *state space exploration tool* that takes in this specification to generate the coherence protocol.

The advantage of this methodology is that the specification of the coherence protocol as well as the interface to the user logic can be changed easily and a new state machine, with the modified specifications, can be automatically generated using the state space exploration tool.

1.2.3 High-level architecture

CCKit distinguishes between the case where a cache line is homed on the FPGA or on the CPU. In neither case does CCKit actually cache the line itself – this is left as a choice to the application logic. Instead, CCKit abstracts the complexity of the protocol, timing, and state machine maintenance and presents a simplified interface to the FPGA application. In the FPGA-homed case, a DC component maintains the directory information for a line, including the local protocol state and the state it believes the line to be in on the CPU.

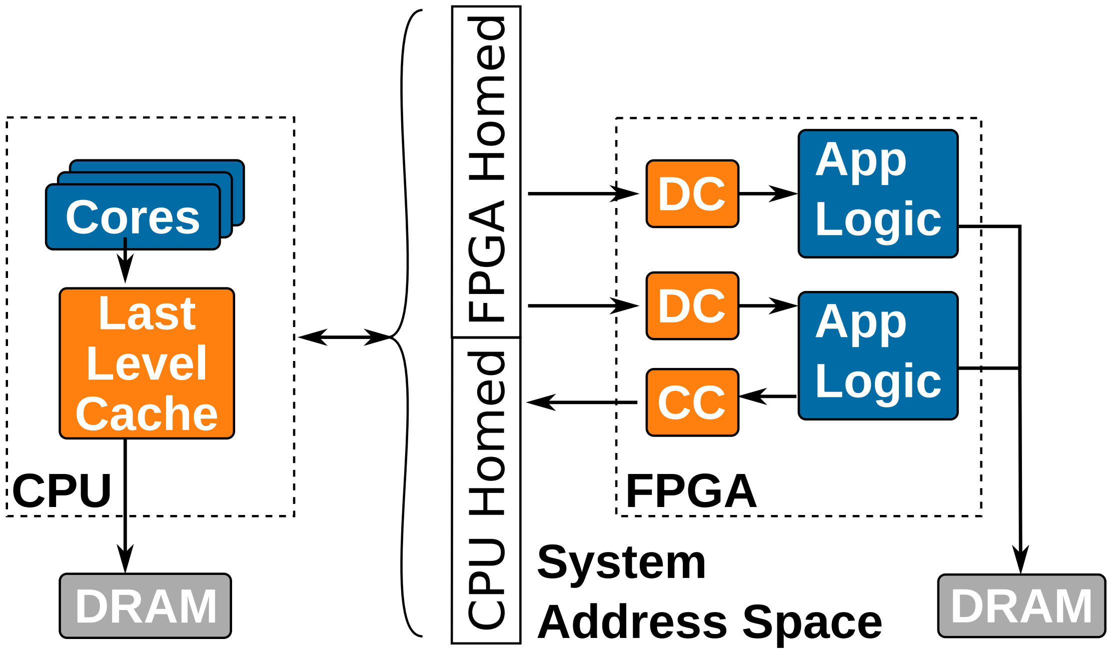


Figure 1.1: *CCKit architecture.*

The CPU-homed case is handled similarly by a Cache Controller (CC) component on the FPGA. Since CCKit does not perform any caching *per se*, the CC is much simpler than the DC, and is to some extent a subset of its functionality. We therefore concentrate on the DC in this thesis; all applications in chapter 9 use the DC.

Figure 1.1 shows the high-level architecture. Each DC or CC is responsible for a different region of the physical address space. By varying the number of units, performance can be traded off against FPGA resources and flexibility (since different units can be configured with variants of the protocol end-point) This mirrors the behavior of a CPU LLC, except that in a CPU the controllers' parameters are hardwired (the only "application" to be supported is the cache itself) thus it makes sense to completely integrate all functionality into the LLC controller.

1.2.4 FPGA-side interface

All request and response messages originating on the CPU and referring to FPGA-homed coherent addresses are routed to a DC, which tracks the state of all lines held on the CPU and initiates and responds to MOESI transactions (e.g. upgrade/downgrade) on behalf of the rest of the FPGA. The interface to the remainder of the FPGA logic is much simpler: An Advanced eXtensible Interface (AXI) interface for reads and writes (to service upgrade and downgrade requests, respectively), plus a request-acknowledge interface allowing FPGA logic to trigger a clean (write-back without invalidate) or clean+invalidate operation on the CPU's LLC. In the simplest implementation, the AXI interface can be connected directly to the FPGA-side DRAM controllers to provide coherent read/write access to FPGA-side DRAM from the CPU. Most non-trivial applications will instantiate their own logic between these two components to interact with the coherent interconnect (chapter 9).

The request-acknowledge interface to FPGA application and the directory protocol are generated from a machine-readable specification and can be configured easily. This can be used to tailor the protocol and interface. Using CCKit, an FPGA programmer does not have to track the state of individual cache lines and can rely on the messages observed at CCKit's interface to infer certain guarantees. For example, an application observing an AXI read request for a cache line can infer that the cache line is invalid in the CPU's cache and the FPGA memory has the most up-to-date copy. Similarly, an AXI write request, when observed, guarantees the cache line is either Invalid or Shared in the CPU's cache but never Exclusive or Modified. Applications that do not require interacting with the request-acknowledge interface can rely on these guarantees provided by the AXI interface. For interactions with the request-acknowledge interface, in the simplest version of the protocol, applications can issue clean and clean-invalidate requests for cache lines and rely on messages observed in both acknowledge and AXI channels to perform what is required (subsection 9.6.3).

Changing the directory protocol can provide additional guarantees for the application, simplifying its design. For example, the application described in subsection 9.6.2 requires a cache line to be invalid in the CPU's cache until the application modifies it and releases control. For this, we generated a version of the protocol where the application can lock a cache line upon performing a clean or clean-invalidate, and we added an additional interface channel for applications to unlock cache lines. This locking mechanism prevents the CPU from upgrading a cache line thereby guaranteeing its state when the request is acknowledged. It also guarantees that the FPGA memory has the most up-to-date value. The cache line remains in this state until the application relinquishes control using an unlock message issued to the DC. This simplifies the application design by moving a part of its state machine into the DC.

By having a more flexible message format at the request-acknowledge interface can reduce the state that needs to be maintained by applications by packing information that would be required to handle request-acknowledge transactions, as part of the request. For example, if the application needs to store and retrieve a certain (nominal) value when handling request-acknowledge transactions, the value can be embedded into the request and received as part of the acknowledgment.

1.2.5 CPU-side interface

The CCKit interface to software is, compared to FPGA user logic, relatively simple: coherence is mostly transparent to software on the CPU, and what explicit cache operations there are on the CPU (flush, invalidate, etc.) simply translate into coherence messages.

The main functionality CCKit exposes to software is the ability to map part of the physical address space serviced by DCs into the virtual address space of a process. We support this in Linux via a kernel module which creates a device for the FPGA's memory space. This can be mapped into a process address space using `mmap()`.

While the MMU can achieve this trivially, assumptions in the Linux memory management implementation introduce challenges. In CCKit we want super-page mappings for efficiency, and because fine-grained translation does not really benefit FPGA applications. We also want the page mappings to be set up eagerly in advance to avoid, for example, prefetch hints from the CPU being ignored by the cache controller because the prefetch would cause a page fault. Unfortunately, both the `MAP_HUGETLB` flag to `mmap` to request large pages and the `MAP_POPULATE` flag to create mappings in advance only apply to memory that the kernel believes to be RAM [BK22], which is not the case here. Essentially Linux so far has no notion of memory that should be treated as RAM for mapping into virtual address space, but not for anything else. For this reason, CCKit bypasses the standard in-kernel interfaces for page mapping for now. The design of better (CPU-side) OS support for heterogeneous memory is ongoing work.

1.2.6 CCKit acceleration model

The acceleration model in PCIe-based non-coherent CPU-FPGA platforms typically uses the FPGA as offload accelerators. The source data that is to be processed is DMA'd into the FPGA and the results are DMA'd back to the CPU. This requires the source and result pages to be marked as non-cacheable since PCIe protocol does not have a coherence layer.

Coherent CPU-FPGA platforms limits user logic on the FPGA to perfroming load and store operations through a CC, just like cores on the CPU. This limits the observability and interaction of FPGA applications with the coherence protocol and targets to accelerate applications that benefit due to "fine-grained" interaction between CPU and FPGA ([CCF+16]) where both nodes are accessing a shared address space. This model of acceleration is possible using CCKit (see subsection 9.6.2).

A different acceleration model that is made possible by CCKit is where user logic on the FPGA can transparently extend the notion of coherence for software running on the CPU. As will be seen in chapter 2, coherence protocols maintain coherence of each cache line (cache line) independently. That is coherence transactions on one cache line is not affected by coherence messages on another cache line. In CCKit, the FPGA application can make interesting associations between unrelated cache lines by observing coherence transactions on one set of cache lines to cause coherence traffic on a different set of cache lines (an example is shown in subsection 9.6.3). Thus in CCKit's acceleration model, applications on FPGA interact with the coherence protocol through the DC to transparently provide certain coherence and consistency guarantees to software on the CPU.

1.3 Related Work

As discussed in section 1.1, there is growing interest in new models of coherence and applications benefiting from coherence.

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 [ZGK+21]. Similarly, CoNDA demonstrates the benefits of finer-grained coherence, and proposes a more customizable protocol to increase efficiency and performance [BGP+19].

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 [ZWC+20]. 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 [HCW+19] could benefit from customizable cache coherence protocols, including the operators explored in chapter 9.

The movement towards data center disaggregation raises questions on how to handle the additional complexity of new memory tiers. Both POND [LBH+23] and TPP [MWD+23] 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 [CIP+21, LYT+21]. For example, MIND advocates a flexible cache coherence protocol integrated into the network [LYT+21]. Clio argues that customizable, application-accessible coherence is desirable in these systems for limiting coherence overhead [GSL+22].

SmartNICs often employ FPGAs to accelerate common networking tasks such as RPC calls [LXA+21] or RDMA [SWC+20, YHS+23]. These systems provide significant improvement, but the addition of coherence using techniques provided by CCKit could provide added benefit. For example, in Dagger, coherence could allow the use of low-latency synchronization primitives instead of complex application-level interactions [LXA+21]. 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 [YHS+23].

A complete discussion of cache coherence simulators is beyond the scope of this section (see [BKP20] for a more thorough discussion). There is no doubt that simulation tools (e.g. ZSim [SK13], gem5 [BBB+11], or CMP$im [JCLJ08]) are essential for the development of protocols and architectures. However, simulating a real application with these tools is slow, and often simulators trade off architectural fidelity and accuracy for speed. 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 [Ver, EKK+23], making the simulation of complex systems and applications under real workloads nearly impossible.

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.

1.4 Contributions

In this section, we try to address the wider implications of CCKit beyond Enzian. One of the aims of CCKit is to be platform agnostic: the principles of CCKit should be applicable to any symmetric coherent platform that implements a directory-based MOESI like coherence protocol. To achieve this, we take a layered approach when modeling CCKit. We identify the set of services and abstractions each layer has to provide for the layer on top to build upon.

Furthermore, we separate out the coherence protocol modeling and the protocol state machine generation from its implementation. The modeling and state machine generation are platform-agnostic which makes the design principles of CCKit applicable to any symmetric coherent platform, whereas the implementation is tailored to the characteristics of the target platform for maximum performance (Enzian in our case). We also show that it is possible to have a performant implementation of the home-agent on the FPGA even if it operates at a much lower frequency when compared to the CPU.

Finally, we define a simplified and platform-agnostic interface for applications to interact with the coherence protocol. This interface aims to provide high controllability and observability for applications to the coherence protocol while simplifying its interaction. Through this interface we explore both traditional and non-traditional acceleration models that are enabled by CCKit. Thus the contributions of this thesis are as follows.

1. A generic model for directory protocol and a state space exploration tool that generates a platform agnostic directory protocol state machine for CCKit.
2. A customizable, performant and platform specific implementation of CCKit's directory protocol state machine on the FPGA, tailored to Enzian.
3. An abstract interface to the directory protocol for applications to interact with.
4. Exploring traditional and non-traditional acceleration models using CCKit.

1.5 Structure of the Dissertation

Chapter 2 provides an introduction to cache coherence. It shows the baseline system models for different types of coherence protocols like snooping and directory based protocols and extends it to provide the baseline system model for Enzian. It also goes into detail on a number of fundamental design choices that were made before building the coherence stack on the FPGA.

Chapter 4 describes the model of DC protocol that was developed by reverse engineering the CPU's native coherence protocol. In the absence of a formal specification of the CPU's native protocol, this model is used to identify transactions that need to be handled by the DC and build a specification. This chapter introduces *state equations* as a way of specifying the protocol and describes how these state equations can be *solved* to generate a state machine with intermediate states necessary to carry out the DC protocol.

Chapter 5 uses the DC protocol model to identify and specify coherence transactions that would allow the CPU to coherently access the FPGA attached memory. It also provides design choices and an algorithm for a state space exploration tool that solves the specification state equations to generate a DC protocol state machine automatically.

Chapter 6 extends the specification by including the state equations of coherence transactions that are generated by the DC and incorporates them in the protocol state machine.

Chapter 7 further extends the specfication by describing the interface exposed by the protocol state machine to applications. It shows examples of how the interface can be customized for different application requirements and uses the state space exploration tool to automatically generate the state machine.

Chapter 8 describes how the DC protocol state machine is implemented on the FPGA. It shows the design choices made to build a customizable and high performance DC that provides a simplified interface to applications. The interfaces and architecture of different components that make up the DC are described in detail.

Chapter 9 focuses on building applications on top of the directory protocol layer. It evaluates the performance of the DC through a number of benchmarks and showcases different ways in which FPGA user logic can be interact with the coherence protocol through the DC when accelerating applications.

If the reader is interested in building applications using the DC, they can focus on chapter 7 to get an idea of DC's application interface, and chapter 9 to understand how the example applications are developed. The reader might also be interested in the figures in Chapter 8 that goes into detail on DC's interface signals.

If the reader is more interested in the specification of the protocol, for they want to formally verify it, or would like to generate a new protocol state machine, they can focus on chapters 5, 6 and 7.

2 Primer on Cache Coherence

2.1 Introduction

This chapter summarizes cache coherence from the point of view of Enzian. The description given here is by no means comprehensive. Readers can refer to [NSHW20b] for a more elaborate and formal discussion.

The structure of this chapter is as follows.

1. Section 2.2 describes a baseline system without caches and how there is no problem of incoherence in such a system.
2. Section 2.3 describes a baseline system with caches and shows the problem of incoherence. It also details how the problem of incoherence is solved using the coherence protocol. This chapter focuses on snooping based coherence protocol and its disadvantages.
3. Section 2.4 shows the baseline system for a directory based protocol and how it solves problems that were present in a snooping based mechanism.
4. Section 2.5 details a baseline NUMA system with directory based coherence protocol. The difference between symmetric and asymmetric NUMA systems is also discussed.

5. Section 3.3 shows the baseline system model for Enzian and introduces its various components. The layered approach to building a coherence stack on the FPGA is also introduced here.

6. Section 3.2.2 describes the fundamental design choices that were made before building the coherence stack.

2.2 Baseline Multicore System without Caches

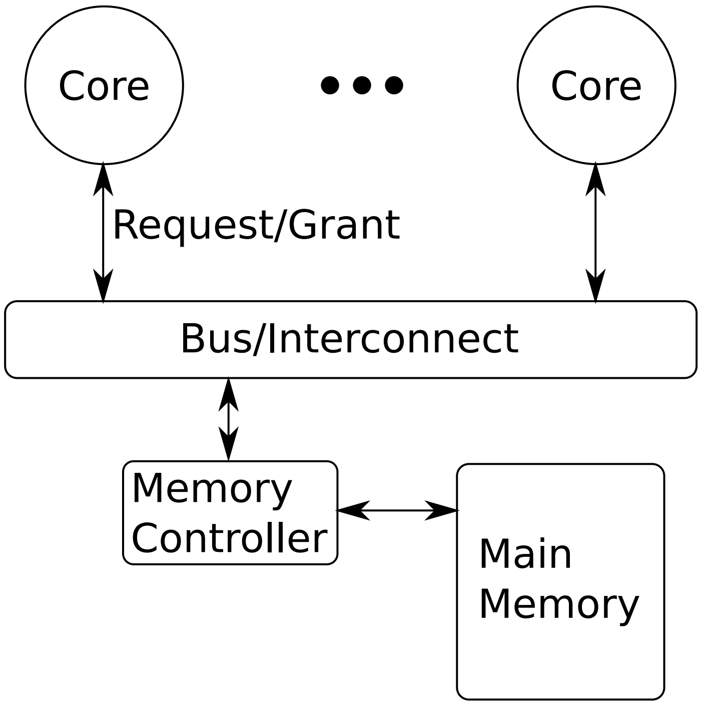


Figure 2.1: *Baseline model for a multicore system without caches: Multiple cores interact with the main-memory controller through a shared bus.*

Figure 2.1 shows a simplified multicore system model without any caches. In this model, each core can make a load or store request to the shared bus. The bus uses an arbitration scheme to choose one core and grant access to main memory. The core can then perform its load or store operation directly on the main memory through the memory controller.

This system offers two advantages: first, only one core can write to a memory location at any given time. Second, there is only one copy of the data for a memory location and the most up-to-date data for the memory location resides in the main memory. The drawback of such a system without caches is the quick exhaustion of memory bandwidth by the processor load [TSS88]. To overcome this drawback caches were introduced.

2.3 Baseline Multicore System with Caches

The baseline system model with caches is described in detail in [NSHW20b] and shown in Figure 2.2. In this system, each core has a private level 1 (L1) cache that is accessible through its CC. The cache controller for all cores are connected to a shared bus or interconnect. Then there can be multiple levels of shared caches before the main memory, though for the sake of simplicity, only one is shown in Figure 2.2 called the LLC. The LLC and main memory are both accessible through the LLC/Memory controller. The difference between L1 and LLC caches are that L1 is private to each core whereas LLC is shared among all cores. In this memory hierarchy as the levels increase, the memory access latency also increases with the main memory having the highest access latency. Although this model does not encompass *all* the different flavors of caching systems that exist in modern multicore systems, it is sufficiently representative to introduce the entities that are present in a multicore system with caches.

Whenever a core wants to read a memory location, it issues a *load* request to its L1 CC. The CC then fetches a copy of the memory location from the memory hierarchy and places it in the private L1 cache of that core. The value can then be loaded into a register in the core from the L1 cache. If a core wants to write to a memory location, the core issues a *store* request to its L1 CC, which then fetches the memory location into its private L1 cache before updating it with the store value from the core. Since any update by a core happens in the private L1 cache of that core, it can lead to the problem of incoherence.

2.3.1 Problem of incoherence

Consider two cores core-1 and core-2 with their private L1 caches. Initially core-1 and core-2 both loads the value of a memory location into their private caches. If core-1 modifies its private copy of the memory location in its L1 cache and if this change is not propagated to the copy on core-2, both cores have different values for the same memory location. Furthermore, the main memory also has a copy which can be potentially stale. This is the problem of incoherence which arises in systems with caches.

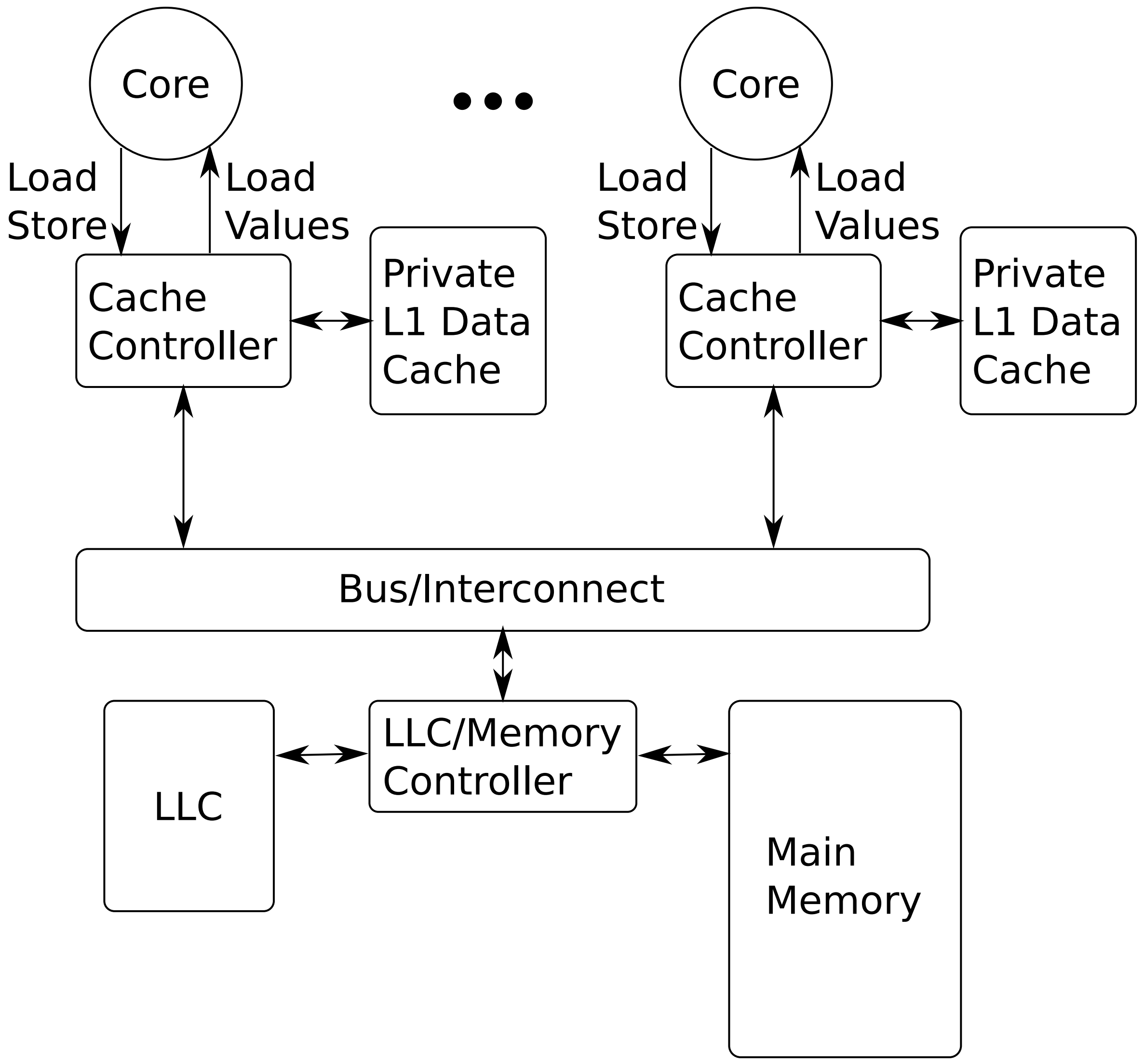


Figure 2.2: *Baseline model for multicore system with caches: Each core has a private cache accessible through a cache controller. Cache controllers communicate with each other and memory controller through a shared bus. The shared bus allows for a snooping based coherence mechanism.*

2.3.2 Shared memory consistency

A second issue with the correctness of memory accesses in a multicore system with caches is shared memory consistency. When multiple threads are accessing a shared memory simultaneously, there can be many possible ways how the memory is accessed. Even within a a single thread, modern ISAs allow multiple possible interleavings of instructions which can change how the memory gets accessed. In order to account for this variety when writing programs, a programmer needs to know what sort of memory accesses can be expected from a system. A memory consistency model defines the correct shared memory behavior by providing rules about correctness of memory accesses. It separates out all possible correct executions from incorrect alternatives for the programmer. There are different consistency models and executions that are correct in one model might be incorrect in another. [NSHW20b] describes in detail different consistency models such as *Sequential Consisteny (SC)*, *Total Store Order (TSO)* and *relaxed memory models*.

2.3.3 Coherence vs consistency

Coherence plays an important role in supporting the consistency model by making the caches functionally invisible in a shared memory system. This means the caching hierarchy need not be considered when defining rules of consistency.

Given a memory location that is being accessed by multiple cores, consistency refers to the order in which the cores get access to the memory location and coherence ensures that irrespective of which core gets access first, both cores see the most up-to-date value for that memory location. For example consider a memory location that serves as a lock which is being accessed atomically by two cores, core-1 and core-2. The order of access can be core-1 followed by core-2, or vice versa. Depending on the consistency model, one or both these access patterns can be valid. Coherence on the other hand, ensures that when one core atomically modifies the lock in its cache, the other core observes that the lock is taken irrespective of the order of access.

2.3.4 Coherence invariants

The problem of incoherence can be solved by maintaining two invariants namely: Single Writer Multiple Reader (SWMR) and data-value invariant. These coherence invariants are maintained by *coherence controllers* in the system such as CCs and LLC/Memory controllers using a protocol called coherence protocol. The description of these invariants are as follows.

SWMR invariant: For a memory location A, at a given time, there exists only one core that may write to A (and can also read it) or a number of cores that may only read A. That is, when a core wants to modify a memory location, only the private cache of that core can have a copy and the rest of the cores cannot have a copy. When reading a memory location, all cores that want to read can have a read only copy of the memory location in their private L1 caches.

This invariant is used to define an *epoch* in the lifetime of a memory location. The lifetime of a memory location can be split into a number of *epochs* where in each epoch there is atmost one core with read-write access to the memory location, or few cores (potentially zero) have read-only access.

Data-value invariant: The value of the memory location at the start of an epoch is the value of the memory location at the end of its last read-write epoch. To illustrate this, consider a core (core-1) that has a modified copy of a memory location in its L1 cache and the contents of the main memory for this memory location is stale. When core-2 wants to read a copy of the memory location into its L1 cache, the data obtained should be the most up-to-date copy from the private cache of core-1 and not the stale copy from main memory.

2.3.5 Cache line

Although cores can load and store data at the granularity of bytes, the data transfer between caches and main memory happen at a much larger granularity called cache line. This is done to offset the higher memory access latency as the levels in memory hierarchy increases. Typically the size of a cache line is 64 Bytes but in Enzian it is 128 Bytes. Coherence is maintained in the granularity of a cache line and not individual bytes. Thus the "memory locations" discussed previously are cache lines and not individual byte addresses.

2.3.6 Maintaining coherence invariants

As seen in subsection 2.3.4 coherence invariants are maintained by coherence controllers such as the cache controller and LLC/memory controller. These coherence controllers maintain *states* of its copy of the cache line. Whenever coherence invariants have to be maintained, these coherence controllers exchange *coherence messages* pertaining to the cache line. The protocol of communication that is to be followed by the coherence controllers is called the coherence protocol.

To illustrate this, let us look at a simple coherence protocol based on the baseline model shown in Figure 2.2 for a *cache line A* with two cores (core-1 and core-2). Initially both

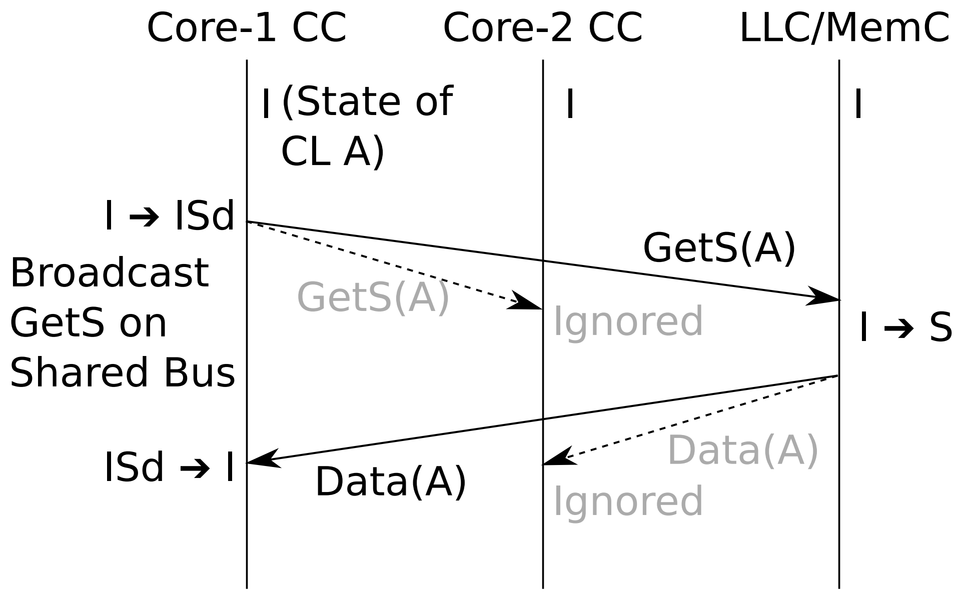


Figure 2.3: *Coherence transaction where core-1 reads cache line A in shared state: Coherence controllers track (intermediate and stable) states and exchange coherence messages to participate in the coherence protocol.*

cores do not have a copy of the cache line in their private caches. That is the state of *cache line A* as maintained by the L1 CC of both cores is *Invalid* (Invalid (I)). Thus the only copy of *cache line A* resides in the main memory. The LLC/memory controller also maintains a state of this cache line. From the point of view of the LLC/memory controller, none of the cores have a copy of this cache line and so it also has the state of the cache line as *Invalid.* An I in a CC means that cache does not have a copy of the cache line whereas an I in the LLC/Memory controller means that none of the private L1 caches have a copy of the cache line.

If core-1 wants to read a copy of *cache line A*, it makes a load request to its CC. The CC then looks up the state of this cache line in its L1 cache. Since there is no copy of the cache line in its L1 (state of *cache line A* is I in L1 cache of core-1), the CC places a Get-Shared (GetS) coherence request for *cache line A* on the shared bus or interconnect as seen in Figure 2.3. The GetS request implies that the cache controller wants a read-only (Shared) copy of the cache line. Once the GetS request has been issued, core-1's CC modifies the state of this cache line in its L1 cache from I to an *intermediate state* ISd, indicating that it is waiting for data ("d" in ISd) before transitioning from I to Shared (S).

Since the bus is shared among all coherence controllers, the GetS request gets broadcasted to all coherence controllers and can be observed by both the CC of core-2 as well as the LLC of the memory controller. The CC of core-2 identifies that it does not have a copy of the cache line and ignores the coherence request. The LLC/memory controller on the other hand knows from its internal state that none of the cores have a copy of the cache line and that the most up-to-date copy resides in the main memory. Thus the LLC/memory controller reads the main memory and places the cache line data on the shared bus as a response to the GetS request from core-1 CC. The LLC/memory controller can also infer that at least one of the cores now has a valid read-only copy of the cache line and so it transitions its state of *cache line A* from I to S. A S state in the LLC/memory controller implies that at least one of the cores has a private read-only copy of the cache line.

With the data for *cache line A* on the bus, both core-1 and core-2's CCs can observe it. Core-2 from its internal state knows that it is not expecting any data for *cache line A* whereas core-1 from its internal state (ISd) knows that it is waiting for *cache line A* data. Core-1 puts the data into its L1 cache and then updates the internal state for *cache line A* from ISd to S. This *coherence transaction* is shown in Figure 2.3.

Coherence transactions are a chain of coherence events towards a common goal. For example, the chain of coherence messages and memory events that provide shared access for a cache line to a coherence controller. Both coherence messages and memory events are together referred to as *coherence events* in this Thesis. Each coherence transaction is either initiated by a request or a response event and subsequent events in the chain of a transaction are all responses i.e. there can be at most one request event per coherence transaction.

From the perspective of maintaining coherence invariants for this coherence transaction, SWMR is maintained because there are no writers and core-1 is the only reader. The data-value invariant is also maintained because core-1 gets the most up-to-date copy of the cache line from the main memory. Thus both coherence invariants are maintained for this coherence transaction.

Next lets look at the coherence transactions involved when core-1 wants to write to *cache line A* instead of reading it. The coherence transaction is shown in Figure 2.4. Initially all three coherence controllers have the state of *cache line A* as I. The CC of core-1 broadcasts a Get-Modified (GetM) coherence request to all coherence controllers and transitions from I to an intermediate state IMd, indicating that it is waiting for data.

When core-2's CC observes this message, it has to invalidate its copy of *cache line A* in

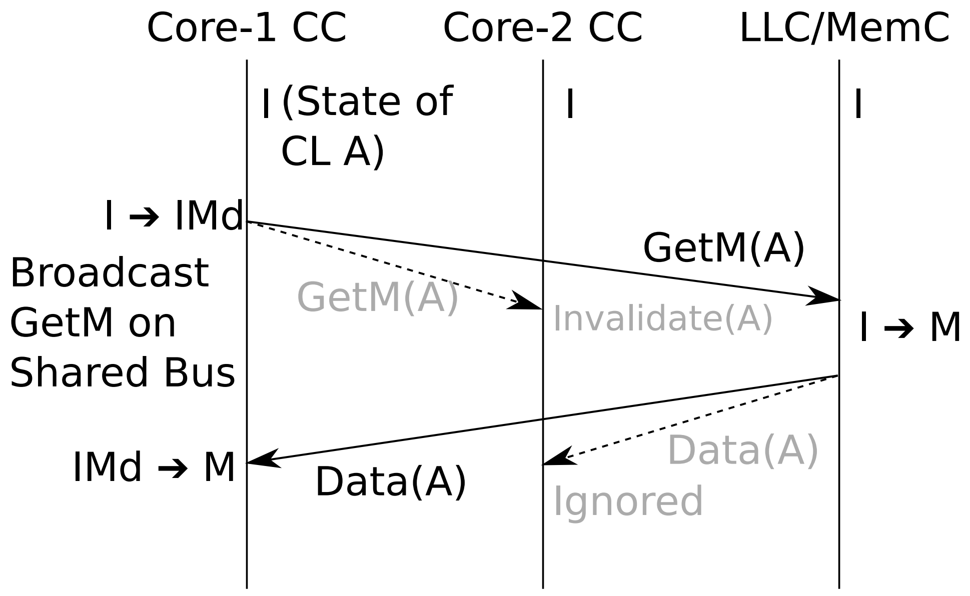


Figure 2.4: *Coherence transaction where core-1 writes cache line A and core-2 does not have a copy.*

order to maintain SWMR. Since core-2's cache does not have a copy of the cache line, the coherence request can be safely ignored by core-2's CC. But the LLC/memory controller from its internal state knows that none of the cores have a copy of the cache line and it responds to the GetM request with the most up-to-date copy of the data from the main memory. Once the data is put on the bus, the LLC/ memory controller updates its internal state of *cache line A* to Modified (M) indicating that there is a modified copy in one of the private caches.

Once core-1 observes the data on the shared bus, it moves the data into its private cache, modifies the data and updates its state from IMd to M. SWMR is maintained because when core-1 issues a GetM request, core-2 tries to invalidate its copy thereby ensuring that only core-1 would have a copy of *cache line A*, the data-value invariant is also met because the LLC/ memory controller responds with most up-to-date data from the memory. This coherence transaction is shown in Figure 2.4.

Finally, lets look at the coherence transaction where core-1 has a modified copy of *cache line A*, and core-2 wants to modify it. Initially core-1's cache controller would have the cache line in M, core-2's CC would have the cache line in I (to maintain SWMR) and the LLC/memory controller also has the cache line in M state.

Core-2 would broadcast the GetM message on the shared bus. When the LLC/memory

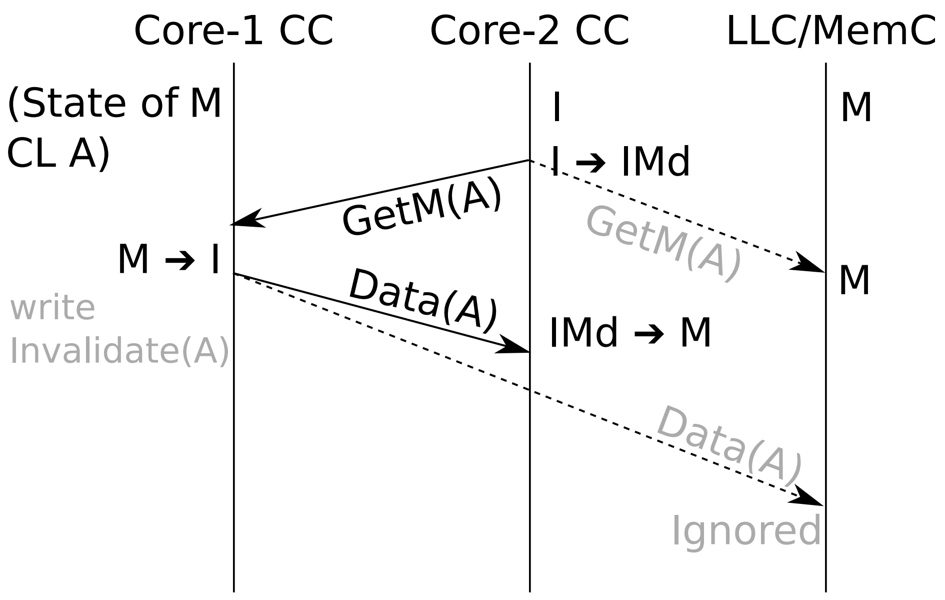


Figure 2.5: *Coherence transaction where core-2 writes cache line A after core-1. core-1 has to invalidate its copy and provide the most up-to-date data to core-2 in order to maintain coherence invariants.*

controller observes this request, from its internal state, it knows that one of the L1 caches has the most up-to-date copy and the contents of main memory are stale. So it does not respond to the GetM request. When core-1 observes the GetM, from its internal state (M), it infers that it has the most up-to-date copy and so it has simultaneously respond to the request with data and invalidate its own copy to maintain SWMR. Thus the core-2 gets the most up-to-date copy from the L1 cache of core-1 thereby maintaining data-value invariant. This coherence transaction is shown in Figure 2.5.

Write-invalidate snooping: These examples illustrate a few coherence transactions of the *MSI* coherence protocol (there are 3 stable states for a cache line: modified, shared or invalid). In this protocol, we observed that a write by one core on a cache line will cause all other cores to invalidate their copy of the cache line. Such a protocol is classified as *write-invalidate* protocol. Moreover, the shared bus acts as a point of serialization for coherence messages, and all cores *snoop* the shared bus to take actions and maintain coherence invariants. Thus this protocol is a *write-invalidate, snooping* protocol.

In all these three examples, whether the data gets loaded into the LLC depends on the configuration of the system. In an *inclusive* configuration, if the data is loaded into the private cache, it would also be loaded into the LLC. In an *exclusive* system, only data that is evicted from the private caches would be present in the LLC. Finally we also have a *neither inclusive nor exclusive (NINE)* which is a combination of both.

2.3.7 Coherence protocol design space

From the example coherence transactions shown in subsection 2.4.1, we can observe the following:

- **There are states associated with cache lines**. These states can be further classified into *stable* states such as M, S or I, and *intermediate* states such as ISd, IMd.
- **Different types of coherence controllers maintain different types of cache line states**. For example, the CCs and LLC/memory controllers both maintain state of cache line with different interpretations.
- **Coherence transactions are initiated and terminated by exchanging coherence messages**. All coherence transactions should ensure that the coherence invariants (SWMR and data-value invariants) are maintained.
- It is also to be noted that coherence transactions on one cache line is not affected by coherence messages on a different cache line. In other words, the **cache lines are mutually independent from the coherence protocol point of view**.
- Next we have the **coherence protocol which identifies all possible coherence transactions between coherence controllers**. There are different types of protocols such as MSI, MESI, MOESI and MESIF with their own set of transactions and optimizations.
- We have also seen that the **protocol can be different for different coherence controllers**. For example, the protocol for a CC is different from the protocol for a LLC/memory controller.
- **Protocol specification** - The protocol specification provides all possible transactions in the coherence protocol. The specification of the CC and LLC/memory controller protocols is together form the protocol specification. The same type of protocol (e.g. MOESI) can have different proprietary specifications.

2.3.8 MSI, MESI, MOESI protocol

Here we briefly discuss two variants of MSI protocol namely MESI and MOESI and how they are useful. Typically coherence protocols are named after the *stable* states in which a cache line can be cached. In subsection 2.4.1 we have seen the MSI protocol, named after its three stable states: I when a cache line is not cached in a cache, S when a cache has a read-only copy of a cache line and M when the cache has a read-write copy of a cache line.

In addition to these states, coherence protocols can have other stable states with the aim of optimizing certain pathways in the coherence protocol. An example is the Exclusive (E) state in the MESI protocol. To illustrate the optimization provided by the E state, consider a cache line that is not cached anywhere (i.e. I state in LLC/memory controller) in a system that implements MESI protocol. Whenever a CC issues a Get-Shared request for this cache line, the LLC/memory controller can provide exclusive access instead of shared access, indicating to the CC that it has the only copy of a cache line. If this copy of the cache line is to be modified at a later point, the CC can silently (without any coherence traffic) upgrade the state of the cache line from E to M thereby reducing the latency of this operation.

Similarly, the Owned (O) state expands a MESI protocol into MOESI, which allows for a dirty copy of a cache line to be *shared* (read-only) between multiple caches without requiring the data to be written back to the main memory. Such inter-cache transfers are faster than accessing the data from main memory. More details on these protocols can be found in [NSHW20b].

2.3.9 Scaling: Snooping vs Directory based protocols

Snooping protocols such as the one discussed in subsection 2.4.1 rely on a shared bus to broadcast coherence messages to all coherence controllers in the system. Electrical constraints limit the scalability of such a shared bus and so snooping protocols are not scalable as the number of cores (and coherence controllers) increase. Directory based protocols are designed to overcome the scalability issue.

2.4 Baseline System with Directory Based Protocol

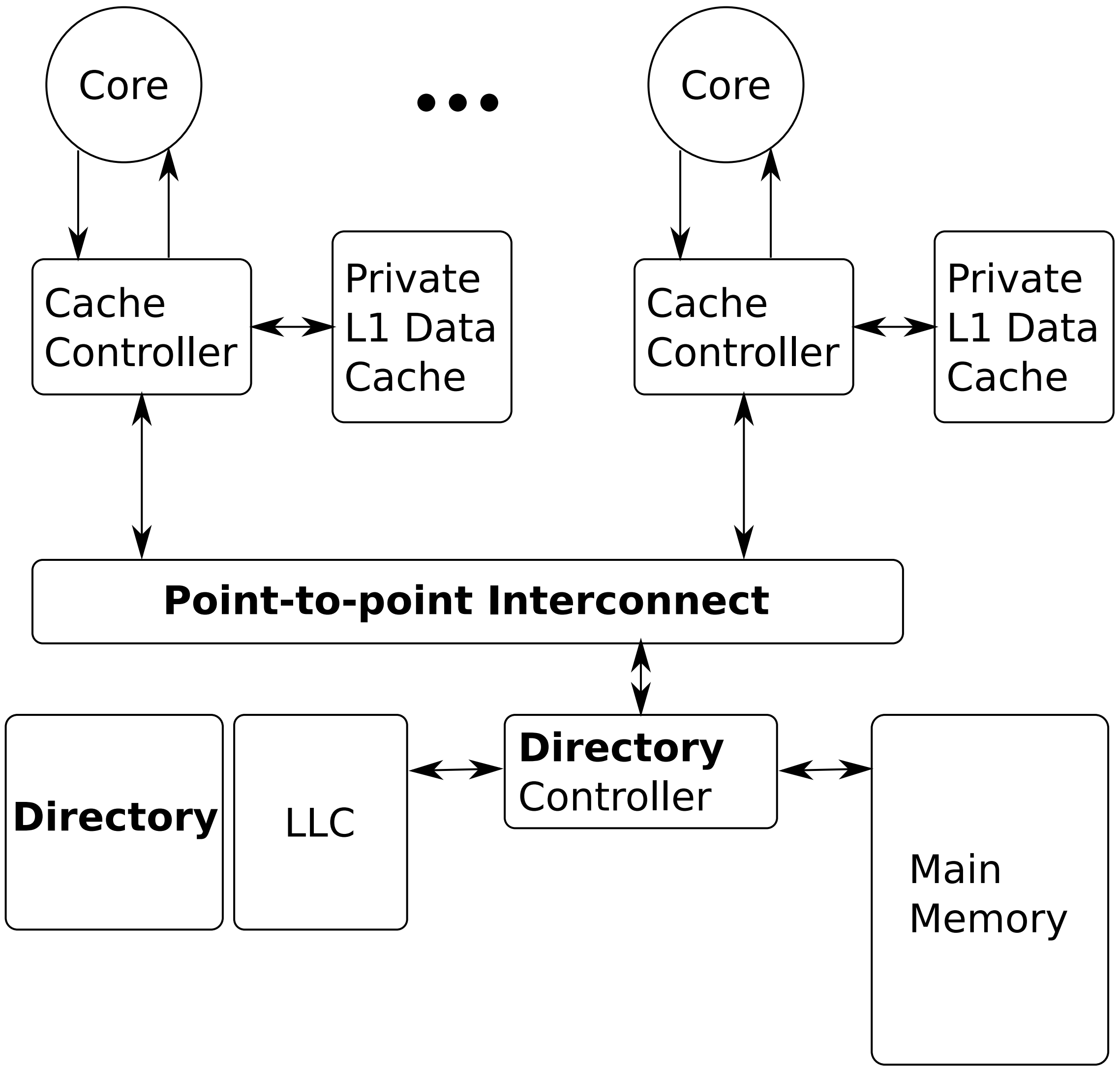


Figure 2.6: *Baseline system model with directory based coherence protocol: The shared bus is replaced with a point-to-point interconnect and a directory is maintained by the directory controller to track global cache line states.*

The baseline system model for a system with directory based coherence protocol is shown in Figure 2.6. There are three main differences compared to the baseline model with snooping protocol shown in Figure 2.2. First, the shared bus is replaced with a *point-to-point* interconnect. Second, the LLC/memory controller is now called a *DC*. Third, a new component called *directory* is added to the system.

Point-to-point interconnect: Unlike a shared bus, the point-to-point interconnect allows for targeted messaging to a specific coherence controller. Messages can be unicast to selected coherence controller without being broadcasted.

DC: The DC is now responsible for maintaining a directory in addition to handling the LLC and main memory.

Directory: The directory holds the global view of the coherence state of each cache line that resides in the main memory. For example, in a multicore system with two cores, core-1 and core-2, if *cache line A* is held in S by core-1's private cache and is not cached by core-2. The directory holds the information shown in Table 2.1.

Cache Line	Core-1	Core-2
A	S	I

Table 2.1: *Directory holds the global view of the coherence state of each cache line.*

The directory is a limited resource and the main memory can have a large number of cache lines. One optimization that is done to reduce the size of directory is to use the directory to keep track of only cache lines that are cached somewhere in the system. A cache line that is not cached anywhere would be in the I state in all private caches. So if an entry is not found for a cache line in the directory, we can assume the state of the cache line in all coherence contollers is I. This would limit the directory to tracking only cache lines that are cached somewhere in the system.

2.4.1 Maintaining coherence invariants in directory based protocol

In a directory based system, coherence requests for a cache line are always unicasted to the DC. The directory controller then looks up the global state of this cache line before deciding course of action. The DC is responsible for maintaining the coherence invariants, reading the main memory and unicasting responses. To illustrate how coherence invariants are maintained in a directory based protocol, let us look at the same examples we had seen in but with a DC.

The system in the example has two cores, core-1 and core-2, connected to their private L1 caches through CCs. The CCs and DC are connected by a point-to-point interconnect. To begin with we have *cache line A* that is not cached anywhere in the system. That is, the state of *cache line A* in core-1 and core-2's L1 cache is I, which is also reflected in the directory.

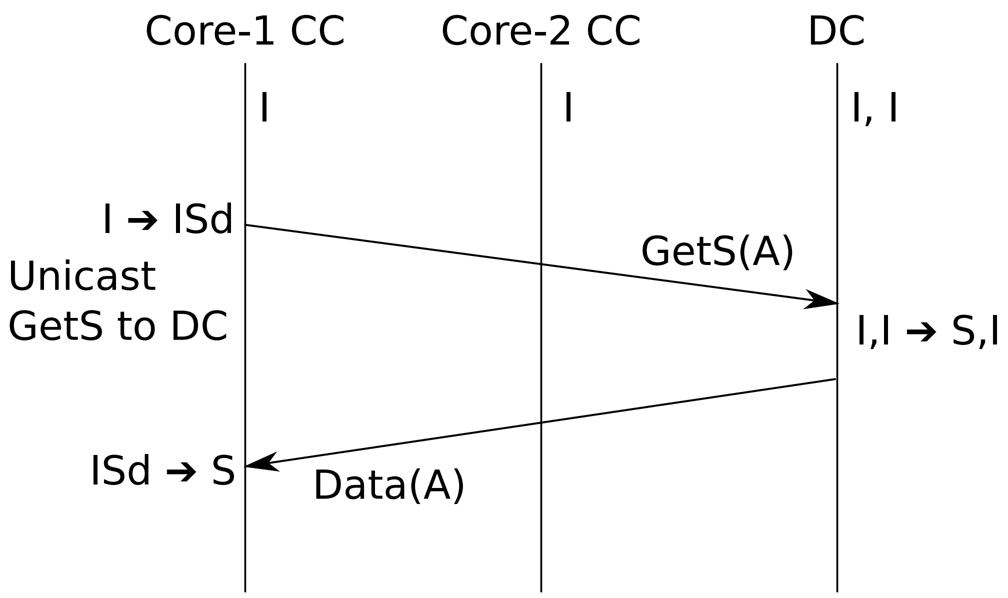


Figure 2.7: *Directory coherence protocol: core-1 reads cache line A in shared state. Coherence requests are unicasted to the directory controller which maintains coherence invariants and provides access to cache lines.*

In the first scenario, core-1 wants to get a shared copy of the cache line. In this case, the CC of core-1 unicasts a GetS request to the DC and updates the internal state of this cache line to intermediate state ISd. Once the DC receives the request, it looks up the directory to identify that the state of this cache line is I in both core-1 and core-2's caches and that the main memory has the most up-to-date value. The DC then reads the memory, unicasts the data to core-1's CC, and updates the global coherence state of the cache line to indicate that core-1 has cache line in S and core-2 has it in I.

Since core-1's CC is waiting for a response, it loads the data into its cache and updates its state of this cache line from ISd to S. Both coherence invariants are maintained in this coherence transaction: there are no writers and only readers so SWMR is maintained, and the DC provides the most up-to-date copy of the cache line to the requesting core thereby maintaining data-value invariant. This coherence transaction is shown in Figure 2.7. It is to be noted that the coherence request is always unicasted by a CC to the DC. The response from the DC is also unicasted to the requesting CC.

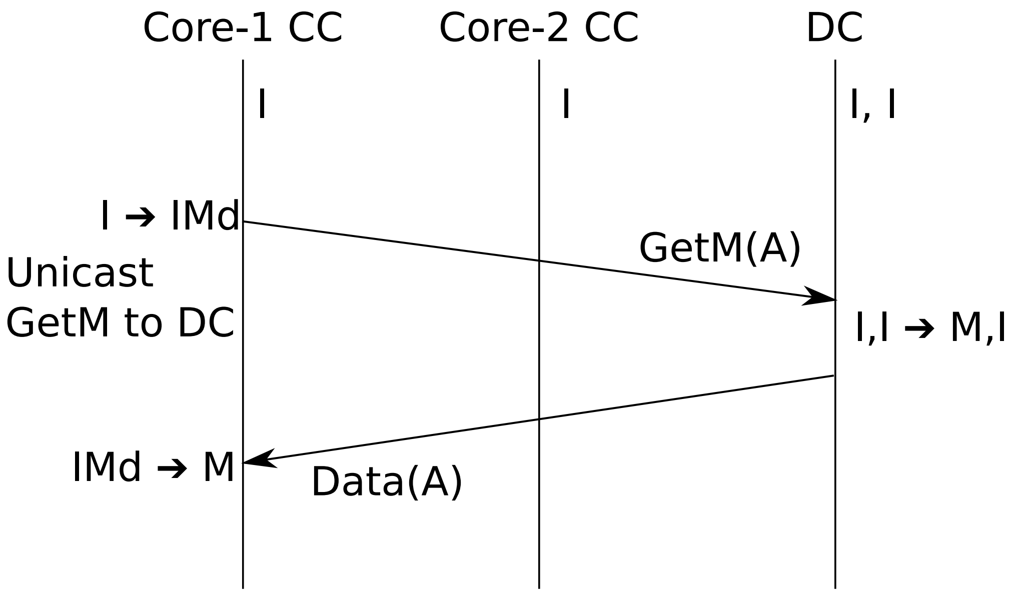


Figure 2.8: *Directory coherence protocol: core-1 writes to cache line A. The directory controller has information that core-2 does not have a copy and allows core-1 to modify the cache line.*

In the second scenario, both core-1 and core-2 do not have copies of *cache line A* and core-1 wants to modify this cache line. Core-1's CC would unicast a GetM request for this cache line to the DC and transition to an intermediate state IMd. When the DC receives the request, it has to ensure that none of the other cores have a copy of this cache line to ensure SWMR. By looking at the global coherence state of this cache line in its directory the DC knows that core-2 does not have a copy and core-1 would be the only writer and the most up-to-date copy residing in the memory would have to be sent as a response. The DC reads the memory and sends the data to the requesting CC and updates the global state of the cache line to indicate that core-1 has a modified copy. Once the data is received by core-1's CC, it gets loaded into the private cache in M state and gets updated in the cache. This coherence transaction is shown in Figure 2.8.

Finally in the third scenario, core-1 has a modified copy of *cache line A* and core-2 wants to write to it. In this scenario, core-2's CC unicasts a GetM request to the DC and transitions to intermediate state IMd. The DC has to ensure that coherence invariants are maintained and from its internal state identifies that core-1 has a modified copy which is also the most up-to-date value. The DC issues a *Forward-GetM* message to core-1 to request it to send the modified data to core-2. The DC also updates the directory indicating that core-1 will have an invalid copy and core-2 gets the modified copy. When core-1's CC receives the Fwd-GetM request, it unicasts the modified data to core-2 and transitions to invalid. This coherence transaction is shown in Figure 2.9.

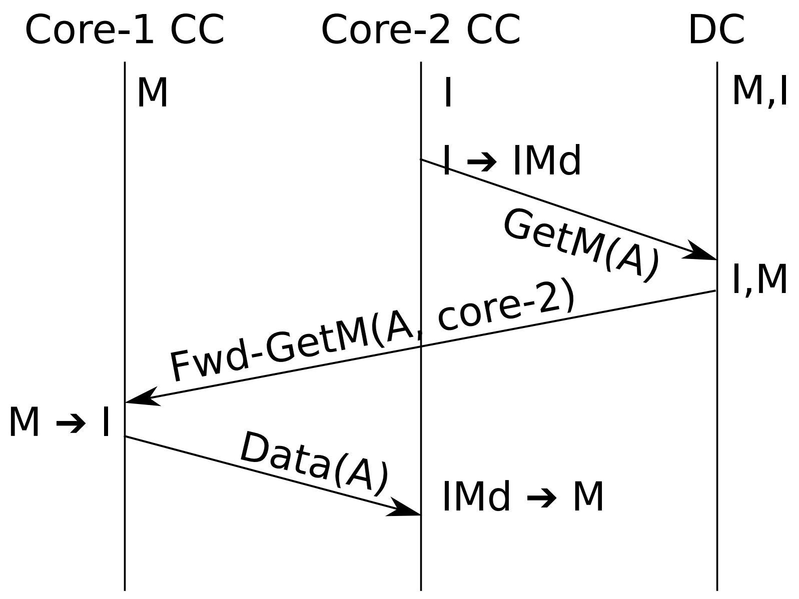


Figure 2.9: *Directory coherence protocol: core-2 writes to cache line A. The directory controller forwards the request to core-1 which has the most up-to-date copy. Core-2 gets data from core-1.*

The forward messages from DC can also be used to evict cache lines from private L1 caches. As a result, the DC can use it to perform directory maintenance and free up directory resources when it gets full. Thus the size of the directory plays an important role in the number of cache lines that can be cached anywhere in the system. In a system with many large caches and a comparatively small directory, the directory would be the performance bottleneck since a forward round-trip transaction would be required to free up directory resources and evict cache lines from caches that are not full. The size of the directory would also limit the caching capacity of the system. Thus an optimal directory would be able to hold the global state of as many cache lines that can be cached in the system. In other words, the size of the directory would track the total caching capacity in the system. This also improves performance as it eliminates round-trip invalidations arising from capacity misses.

Takeaway 2.1. *The directory holds the global states of only cache lines that are cached somewhere in the system. The size of the directory limits the caching capacity of the system* *as well as affects performance due to round-trip invalidations due to capacity misses. An optimal directory size would match the total caching capacity in the system.*

2.5 Directory Based Coherence in NUMA Systems

The baseline system model for a NUMA system with directory based coherence is shown in Figure 2.10. The system has two nodes, with each node connected to its own memory. The two nodes are connected by a *coherent interconnect* that allows the nodes to exchange coherence messages. Each node can access the remote node's memory through the coherent interconnect albeit non-uniformly.

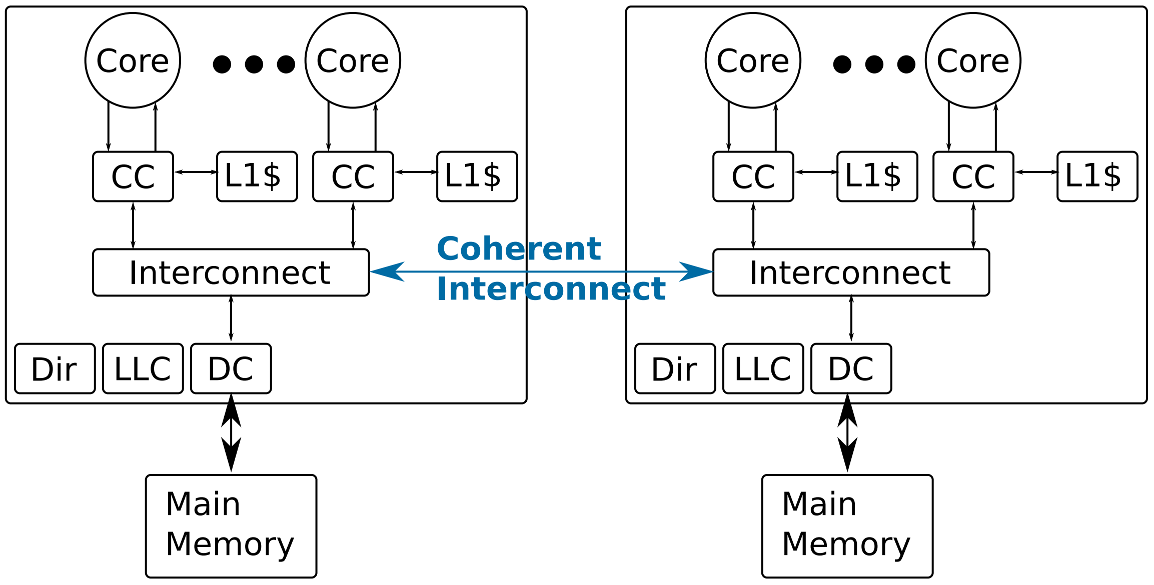


Figure 2.10: *Baseline system model for symmetric NUMA systems: Each node is responsible for maintaining coherence of the memory attached to it.*

Within each node, the coherence protocol can be snooping or directory based but across the nodes it is directory based. The node to which a memory is connected to is the *home node* for that memory. A cache line in a memory *belongs* to the home node. For this cache line any other node is *remote node*. The home node is responsible for maintaining the coherence of all cache lines that belong to it. Coherence invariants for a cache line is maintained by the DC of its home node. Any coherence request for a cache line should be unicasted to the DC of the home node. The directory of the home node keeps track of the global coherence state of only cache lines that belongs to it, and does not track any state of cache lines that belong to remote nodes. For example, in a two-node NUMA system *cache lines A and B* belong to node 1. *cache line A* is in S state in both node-1 and node-2, whereas *cache line B* is in M state in node-1 and I in node-2, then the directory of node-1 would have the information shown in Table 2.2. Since both cache lines belongs to node-1, the home-state (HS) is the state in node-1 and remote state (RS) is the state in node-2

Cache Line	HS	RS
A	S	S
B	M	I

Table 2.2: *Contents of the directory of node-1 based on the example given above for cache lines A, B that belong to it.*

2.5.1 Symmetric vs Asymmetric coherent platforms

We had briefly discussed about platforms implementing symmetric and asymmetric protocols in subsection 1.1.1, here we go a bit deeper in the advantages offered by symmetric protocols over asymmetric.

In a coherent platform with a symmetric protocol, both nodes are equal partners in the coherence protocol. Each node is responsible for maintaining the coherence of the memory attached to it and has a DC to do this. The baseline model of a symmetric system is shown in Figure 2.10. Traditional CPU-CPU NUMA systems are symmetric in nature.

In a coherent platform with asymmetric protocol, one node is responsible for maintaining coherence for memories attached to both nodes. As a result, only the node responsible implements a DC and the other node only implements CCs to interact with the DC. Asymmetric model has been proposed for CPU-Accelerator systems (CPU-FPGA or CPU-GPU etc) with complexity cited as an argument [CXL20b].

The baseline system model for asymmetric NUMA system with two nodes is shown in Figure 2.11. For example, in a CPU-FPGA NUMA system, CPU would be the node on the left which implements a DC whereas FPGA would be the node on the right which does not implement a directory controller but rather only CCs.

The first drawback of such an asymmetric platform is the FPGA cannot maintain coherence of its own memory. Although there is a *bypass* available for non-coherent access, the

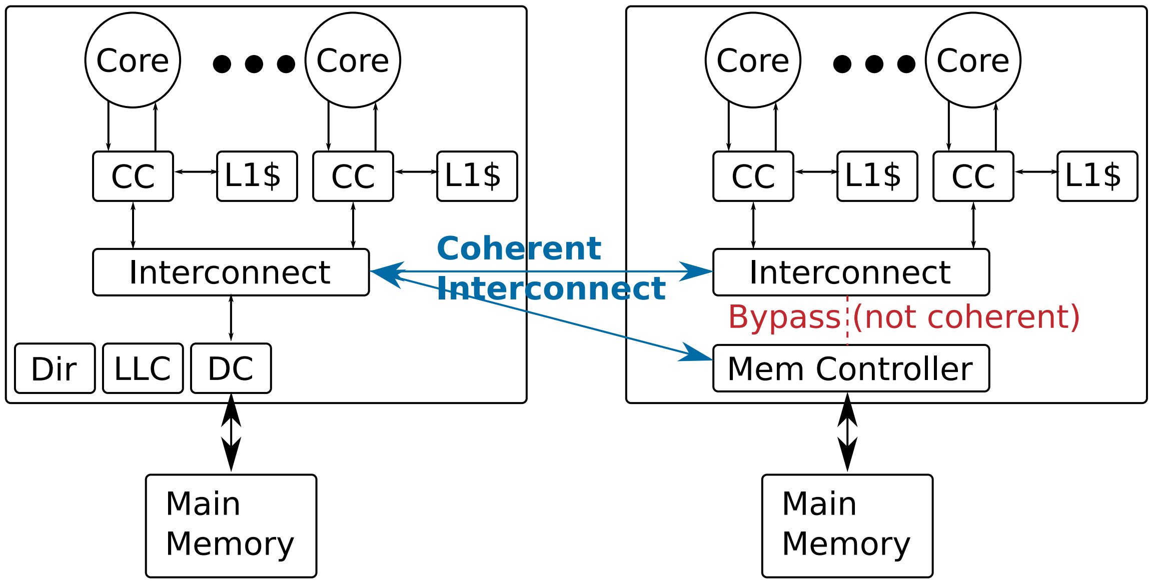


Figure 2.11: *Baseline system model for asymmetric NUMA systems: Only one node maintains coherence of memory attached to both nodes.*

application on the FPGA would always have to make a request to the CPU to be able to access its own memory coherently which increases latency. Secondly, the FPGA loses control over the entire section of the coherence protocol. For example, applications cannot issue messages that belong to the directory protocol such as the forward downgrade messages (e.g. Fwd-GetM message in Figure 2.9) that can cause cache lines to be cleaned or evicted from the remote node's LLC. Thirdly, the FPGA also loses observability over the directory protocol section of the coherence protocol. For example, applications on the FPGA cannot observe any messages that are issued by other coherence controllers to the DC for its own memory. Finally, 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 on cache lines by applications making it difficult to design applications that can be proved to be correct.

3 CCKit Baseline System Model and Enzian

3.1 Introduction

In the previous chapter, we introduced the basics of coherence protocols and introduced simplified system models for different types of multi-core systems. In this chapter we ask ourselves the question: *What would the baseline system model look like for CCKit?*

We develop a baseline system model for CCKit, and look at the design choices made to limit the scope of CCKit. Next, we will define the general characteristics of a platform that is targeted by CCKit, and introduce Enzian [CRS+22], our platform of choice to implement CCKit. Finally we will provide an alternate way of looking at the components of CCKit, which would be useful to understand its acceleration model and how applications can be developed using CCKit.

The structure of this chapter is as follows.

1. Section section 3.2 describes the baseline system model for CCKit, the target platform assumptions, and the scope of this work.

2. Section section 3.3 describes the Enzian platform and its coherent interconnect called Enzian Coherent Interconnect (ECI).

3. Section section 3.4 describes the protocol stack that will be useful to understand application development on the FPGA.

3.2 CCKit Baseline System Model

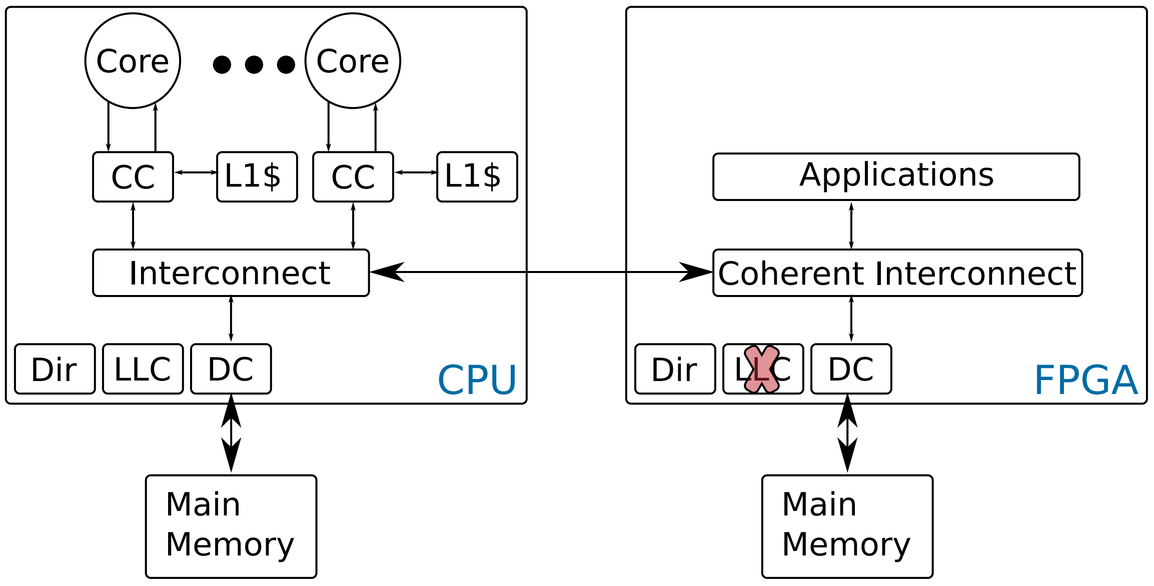


Figure 3.1: *Baseline system model for CCKit: CCKit is implemented on a coherent symmetric NUMA CPU-FPGA heterogeneous platform. Only FPGA-homed scenario (DC) is considered here and CPU-homed case (CC) is not.*

In this section, we will look at CCKit's baseline system model and the assumptions that go into it. This model builds on the baseline system model of symmetric NUMA systems described in section 2.5. We will also discuss the design choices made in this model as well as the reasons and drawbacks of each choice.

Previously in subsection 1.2.3 we discussed the high level architecture of CCKit and its two components namely CC for CPU-homed cache lines and DC for FPGA-homed cache lines. In this thesis we concentrate on the DC and the baseline system model for CCKit is shown in Figure 3.1.

In this model, the CPU and FPGA are connected through a symmetric coherent interconnect where both nodes are equal partners in the coherence protocol. On the FPGA we have a DC that abstracts the complexity of the directory protocol, and state machine maintenance and provides a simplified interface to the application on the FPGA. It also provides coherent access to the FPGA main memory.

The coherent interconnect exposes the coherence protocol to the FPGA by allowing the CPU and FPGA to exchange coherence messages. The coherence protocol in CCKit's baseline model should be symmetric, directory-based, write-invalidate MOESI coherence protocol with no NACKs (no-acknowledgments). Additional assumptions on the rules of interaction between the CPU and DC are described in section 4.3. These simple assumptions cover a wide variety of real-world coherence protocols including those exposed by ECI on Enzian and, hopefully, CXL 3.0.

The assumptions made on the implementation of the coherent interconnect are as follows. First, the interconnect should allow for deadlock-free exchange of coherence messages. Secondly, the interconnect should guarantee the delivery of coherence messages. Thirdly, the interconnect *does not* have to guarantee the ordering of coherence messages.

3.2.1 CCKit target platform assumptions

In the previous section, we briefly discussed CCKit's baseline system model and assumptions made by CCKit on the platforms it targets. In this section, we will delve a bit deeper into these assumptions.

Symmetric coherent CPU-FPGA platforms: CCKit targets symmetric coherent CPU-FPGA platforms. We did not choose asymmetric coherent platforms because the FPGA is not an equal partner in the coherence protocol in such systems. This choice allows us to explore the role of FPGA's in symmetric platforms but comes at the cost of handling the complexities of the coherence protocol on the FPGA.

Directory based protocol: CCKit targets NUMA systems which always implement a directory based protocol instead of snooping protocols as snooping protocols do not scale well. Snooping protocols are used to asymmetrically connect FPGAs to the processing system in MPSoCs which are used mostly in embedded applications. The choice made allows us to target platforms with server grade CPUs and large FPGAs.

Write-invalidate protocols: Write-update protocols, which are alternatives to write-invalidate protocols, are rarely implemented because of their complexity [NSHW20b]. Well-known coherence protocols like MSI, MESI, MOESI, MOSI and MESIF are write invalidate protocols. Thus CCKit targets coherence protocols that are write-invalidate.

MOESI protocols with no NACKs: MOESI protocols are widely used in commercial systems and hence is a nature choice to base CCKit on. Additional assumptions on the rules of interaction between the CPU and DC are described in section 4.3. No-acknowledgments (NACKs) are not widely used in commercial coherence protocols because they can lead to livelock problems [KCA92] and so are not considered here.

Coherent interconnect: CCKit expects the coherent interconnect to be deadlock-free. This is typically achieved by having several virtual channels for different coherence message classes in the interconnect's implementation. CCKit also expects guaranteed delivery of messages. This assumption is nominal because modern interconnects such as ECI and PCIe have layers of infrastructure that check for errors in data exchanged and retry transmission upon failure. Finally, CCKit makes no assumptions on the ordering of coherence messages in the interconnect. Ordering requirements on coherence messages can limit the peformance of interconnects and so high performance interconnects will not provide any ordering guarantees.

3.2.2 CCKit fundamental design choices

In addition to the assumptions we make about the target platfrom, we also made the following design choices to restrict the scope of this work.

Design Choice 3.1. *No LLC would be implemented on the FPGA. The first reason for this design choice is having a caching layer prevents FPGA applications from directly interacting with the coherence protocols; Applications only perform loads and stores on the cache. The second reason is that the FPGA operates at a much lower frequency (order of MHz) compared to the CPU (order of GHz) and caches on the FPGA* might *not be useful. The drawback of this design choice is foregoing any performance benefits in case a cache is useful.*

Design Choice 3.2. *A MESI variant of the protocol would be implemented on the FPGA instead of MOESI. The reason for this design choice is that the O state of MOESI protocol is useful only for inter-cache transfer between the CPU and FPGA (see subsection 2.3.8). Since the FPGA does not have an LLC, this optimization is not needed and a MESI variant would suffice. Alternatively, how a MOESI protocol would look like without an LLC is an interesting question that can be explored in the future. Such a study can identify the drawbacks in the current choice.*

Design Choice 3.3. ***CPU application to FPGA memory:*** *The DC would fully support coherent caching operations from the CPU to the FPGA memory. This is the best case scenario and an alternative would be to support the traditional DMA acceleration model using only coherent non-caching operations.*

Design Choice 3.4. ***FPGA application to FPGA memory:*** *The DC would only support coherent non-caching operations from applications on the FPGA to FPGA's memory. This means FPGA applications cannot cache FPGA homed cache lines but rather read and write to them in a coherent non-caching manner. The alternative choice would be to allow both caching and non-caching operations but, as will be seen later in chapter 9, this design choice does not restrict the capabilities of FPGA applications.*

Design Choice 3.5. ***FPGA application to CPU memory:*** *The DC on the FPGA is not responsible for providing access to CPU's memory. There would be no support on the DC for FPGA applications to access CPU's memory. Coherent access from FPGA applications to the CPU attached memory requires a CC to be implemented on the FPGA and is not in the scope of this work.*

3.3 Enzian

In section 3.2 we had discussed the nature of a platform that on which CCKit can be implemented. Enzian [CRS+22] is one such 2-socket heterogenous server platform. In this section we will look into details of Enzian and its coherent interconnect called ECI.

Overlaying CCKit's system model shown in Figure 3.1 on Enzian, we have two nodes, the first node is a Marvell ThunderX-1 1 CN8890-NT 48-core ARMv8-A CPU running at 2.0 GHz. The second node is a Xilinx VU9P UltraScale+ FPGA [Xil21] that runs at 322MHz. These nodes are connected by the CPU's native coherent interconnect.

The CPU has a snooping based protocol for coherence between the 48-cores within the CPU and exposes a directory based coherence protocol across the two nodes through the coherent interconnect. The CPU has 16MiB of 16-way associative LLC and the private L1 caches are all write-through. The interconnect offers a theoretical bandwidth of 30 GiB/s and the cache lines are 128 Bytes wide. The CPU implements a proprietary variant of MOESI protocol and comes with its own implementation of the DC and directory that provides access to 128 GiB of CPU main memory.

The FPGA is connected to the serial lanes of the CPU's native coherent interconnect through 24 10 Gb/s transceiver lanes and is otherwise a blank slate. These lanes are organized into two 12-lane links with the coherence traffic load is shared between them. This means that although two links are necessary for peak interconnect perfomance, one link is sufficient to run the coherence protocol (at half the performance).

Infrastructure is needed on the FPGA to decode/encode data from the serial lanes and expose interfaces to send and receive coherence messages. This infrastructure is called ECI (Enzian Coherent Interconnect). ECI exposes the coherence protocol at the message level to applications on the FPGA and abstracts all the lower communication layers. Since ECI is their own implementation of all layers of the CPU's native coherent interconnect, the interconnect in Enzian is called ECI.

Upto 1 TiB of main memory can be attached to the the FPGA and current systems use either 512 GiB or 64 GiB of memory. Since, Enzian implements a symmetric protocol, a DC along with its directory is necessary on the FPGA to be able to coherently access this memory.

Thus comparing to CCKit's baseline system model shown in and Figure 3.1, the *coherent interconnect* on the FPGA is ECI. Applications running on both CPU and FPGA can coherently access the FPGA attached memory through ECI and DC. Moreover Enzian implements a symmetric MOESI based protocol which makes it suitable platform to implement and evaluate CCKit.

Note 3.1. *In this thesis, the coherent interconnect on Enzian is refered to as ECI instead of the name given to the CPU's native interconnect. This is because ECI is the Enzian team's own open-sourced implementation of all communication layers of the interconnect. In a twist of fate, the Enzian team is also us wearing a different hat.*

3.3.1 Enzian Coherent Interconnect (ECI)

As mentioned in the previous section, ECI exposes the coherence protocol to the FPGA at the message level. The implementation of ECI on Enzian provides deadlock-free and high performance exchange of coherence messages with guaranteed delivery but no guarantees in ordering of coherence messages. This satisfies CCKit's requirements on coherent interconnects discussed in section 3.2. In this section, we look a bit deeper into how these guarantees are provided by ECI.

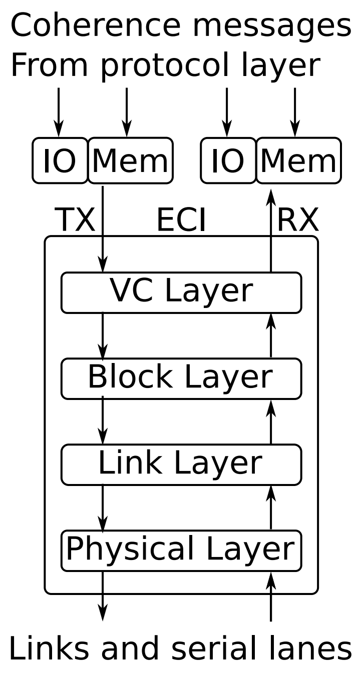


Figure 3.2: *Layers of ECI: ECI allows deadlock free exchange of coherence messages. ECI guarantees delivery of messages but does not guarantee ordering.*

ECI follows a layered approach to provide deadlock-free exchange of coherence messages with guaranteed delivery. Layers of ECI are shown in Figure 3.2 and are described below.

VC layer: Coherence messages are classified into a number message classes by the CPU's native interconnect. Each message class is assigned to one or more VCs to avoid deadlocks during exchange of coherence messages. Each VC can be viewed as a First In First Out (FIFO) that has a credit-based flow control to throttle the rate of sending and receiving coherence messages in each message class.

Each VC is assigned a VC number by the CPU's native interconnect and ECI follows the same numbering scheme. The different VCs and their numbers are shown in Table 3.1. It is to be noted that for message classes with two VC numbers, the odd VC number carries even cache line indices and vice versa.

ECI splits the physical address space in two, namely the IO address space and memory address space. The IO address space is used for configuration registers and is not coherent. The messages for IO address spaces are exchanged through the IO request and response VCs (VC0, VC1). At the application level, these coherence messages are exposed as an Advanced eXtensible Interface Lite (AXI-Lite) interface. The AXI-Lite has a data-bus

Message Classes	VC Number
IO Requests	VC0
IO Responses	VC1
Request with Data	VC2, VC3
Response with Data	VC4, VC5
Request w/o Data	VC6, VC7
Forward w/o Data	VC8, VC9
Response w/o Data	VC10, VC11
Multiplexed coprocessor	VC12
Multicore Debug	VC13

Table 3.1: *ECI message classes and their VC numbers: Coherence messages are classified into message classes, with each message class having a VC to exchange coherence messages. This allows deadlock free exchange of coherence messages over ECI.*

width of 64 bits and each coherence message in the IO address space can only carry 64 bits of data (as opposed to 1024 bits or 128 bytes that can be carried by coherence messages on memory address space).

VCs 2 to 11 are used to exchange coherence messages for the memory address space with the protocol layer that resides above the VC layer. Messages of type requests, responses and forwards have separate VCs. This helps avoid deadlocks, for example if a single VC is required for both requests and responses, then a response that is needed to handle a request might get stuck behind the request in the VC leading to a deadlock. The request and response message classes are further split into ones with cache line data and without. This classification helps separate the control plane from data plane which is useful during implementation of layers above the VC layer.

VCs 12 and 13 are not used for exchanging coherence messages but pertain to co-processor or debug messages. It is not yet known how to use these VCs.

It is to be noted that ECI provides separate sets of VCs for odd and even cache line indices to increase parallelism and improve performance.

ECI block layer: Coherence messages can be of different sizes depending on the type and amount of data present in them. These messages are packed into “blocks” or “ECI frames” that are of fixed size and have a specific format. The data from all VCs are effectively packed into blocks in the ECI block layer. This layer keeps track of blocks using a sequence number and helps in re-transmission of the block when a cyclic redundancy check (CRC) error is detected. This ensures that once a coherence message is pushed into a VC, it is guaranteed to be delivered but might not be delivered in the same order in which it was sent.

ECI link layer: This layer is responsible for implementing the link state machine that brings up the two links and maintain them. The ECI link layer on the FPGA communicates with the link layer of the CPU to ensure the links stays up and working.

ECI physical layer: This is the lower most layer of ECI that is responsible for encoding and transmitting, or decoding and receiving data from serial lanes.

Thus ECI provides the following guarantees to layers that are built above it.

- ECI provides deadlock free exchange of coherence messages.
- ECI guarantees delivery of coherence messages.
- ECI does not guarantee the ordering of coherence messages.
- Delivery of messages by ECI is not instantaneous.
- ECI is optimized for performance.

3.4 CCKit protocol layers

An alternate way of viewing the components of CCKit is in the form of the layers that are stacked on top of each other, where the lower layers provide certain services to the higher layers. This view is useful when reasoning about the acceleration models that are enabled by CCKit. In this section we will look at the three layers of CCKit, and the services offered by each.

We have seen in subsection 1.2.6 that the acceleration model in CCKit is for user logic on the FPGA to extend the notion of coherence to software on the CPU by coherently associating unrelated cache lines. To achieve this, CCKit's components can be viewed in terms of two more layers of protocols built on top of ECI. The protocols are shown in Figure 3.3 and are as follows.

We have already seen the guarantees (a.k.a services) provided by ECI in subsection 3.3.1. The DC protocol layer is built on top of ECI. So in addition to guarantees provided by ECI, the DC protocol layer provides the following guarantees.

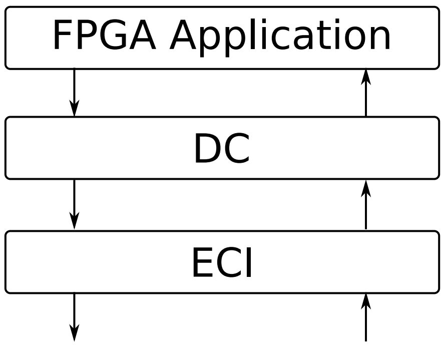


Figure 3.3: *CCKit protocol layers: Each layer provides certain guarantees and is built on the guarantees provided by the layer below it.*

- DC guarantees that coherence invariants are maintained at the granularity of a cache line by maintaining their coherence states.
- DC ensures coherence transactions on different cache lines are mutually independent.
- DC accounts for conflict conditions that can arise due to reordering of coherence messages by ECI and latency of message transfer in ECI.
- DC is deadlock free.
- DC is optimized for performance in critical paths.

Finally we have the application protocol built on top of DC layer. This layer changes from application to application without any modifications to either DC or ECI layers. The application protocol can make associations and guarantee invariants across different cache lines. A simple example of such invariant is if *cache line A* is cached in CPU's cache then *cache line B* should not be cached in CPU's cache. The considerations when building application layer protocol are as follows.

- The application layer does not have to maintain coherence states of cache lines and and rely on the DC to ensure coherence invariants are maintained.
- The application layer can also rely on the DC to keep coherent transactions on different cache lines mutually independent. This means, the application can stall coherence transaction on one cache line to initiate a different coherent transaction on another cache line and wait for it to finish before continuing with the stalled transaction.

- The application must adhere to the specification of DC's interface when interacting with the DC to avoid deadlocks.
- The application protocol itself must be deadlock free.
- The application can be optimized for performance.

More information on building the application protocol on top of DC protocol can be found in section 7.8.

3.4.1 Two components of a protocol layer

Both DC and application protocol layers consist of two components, namely the *protocol state machine* and *its implementation*. The protocol state machine handles coherence events by identifying the state transitions and actions to be performed to comply with the protocol. It should be deadlock free and any deadlocks in the protocol state machine is referred to as *protocol deadlocks*. The protocol state machine should also be optimized for performance.

The second component is the *implementation* of the state machine on the FPGA. The protocol state machine does not take into account resource availability and it is the responsibility of the implementation to faithfully recreate the protocol state machine in spite of having limited resources. The implementation should also be performant and avoid deadlocks that arise due to resource conflicts. These deadlocks are referred to as *resource* deadlocks.

Having protocol state machine that is optimized for performance and has no protocol deadlocks is necessary for the implementation itself to be deadlock free and performant but it is not enough. The implementation also should be performant and free of resource deadlocks.

Reason for protocol deadlocks: A protocol deadlock can occur if the protocol state machine waits for a coherence event that it might never receive. One reason this might happen is if the protocol specification is buggy. The second reason protocol deadlocks can occur is when handling a coherence event by the state machine is dependent on other events being received. For example, the state machine waits for a response before handling a new coherence request. In this case, a deadlock can occur if the response is behind the request in a queue which necessitates that the request be handled for the response to be received by the state machine.

3.4.2 DC protocol state machine design space on Enzian

The DC protocol state machine handle coherence requests from coherence controllers, maintain coherence invariants and provides coherent access to the FPGA attached memory. In order to build the directory protocol state machine, we need to explore the protocol design space as we did in subsection 2.3.7 and answer the following questions.

- **State of cache line in DC:** What are the stable and transient states that are required to maintain coherence for FPGA homed cache lines?
- **Coherence transactions:** We have a list of coherence messages that were provided to us by Cavium but do not have any information on the allowed coherence transactions. What are the coherence transactions that can be initated by CPU CCs and FPGA applications in the system that would have to be handled by the DC?
- **Coherence protocol:** Would a full MOESI protocol need to be implemented on the DC?
- **Protocol specification:** There is no specification of the directory protocol and it would have to be reverse engineered.
- The CPU's native coherence protocol was never designed to inter-operate with an FPGA, how do you make the CPU believe it is talking to another CPU and not an FPGA (even if FPGA operates at a much lower frequency).

3.5 Summary

In conclusion, we have developed a baseline system model for CCKit. We have also identified the characteristics of a platform that is targeted by CCKit. Next, we looked at how components of CCKit can be viewed as layers of a protocol stack that would help us understand what role the FPGA can play in its interaction with the coherence protocol. Finally, we also defined the services that would have to be provided by each layer in the protocol stack. In the next chapter, we will model the interaction between the CPU and the FPGA's DC.

4 Directory Protocol Modeling and Specification

4.1 Introduction

Previously in Figure 3.1 we have seen the baseline model of a symmetric coherent CPU-FPGA system. In this system, the CPU's LLC exchanges coherence messages with the DC through a coherent interconnect (ECI in case of Enzian). The DC, in turn, implements the directory protocol and provides coherent access to the FPGA attached memory. Before we consider how the directory protocol will be implemented, we need to understand what is the model of interaction different components of this system and the DC. Towards this direction, we first ask the question *what is the model of interaction between, specifically, the CPU and the DC in this symmetric system?* The directory protocol specification should provide an answer to this question, provided it exists. In our case, we did not have a formal specification of the directory protocol which leads us to our next question *How do we model and specify a directory protocol?* This is the topic of this chapter.

Before delving into modeling the directory protocol, we had mentioned in section 3.4 the guarantees that the directory protocol should provide: First, it should guarantee that coherence invariants are maintained at the granularity of a cache line. Second, coherence transactions on different cache lines should be mutually independent. Third, the it should account for conflicts that can arise due to the highly concurrent nature of the protocol and interconnect. Last, it should be deadlock free and optimized for performance. These guarantees will decide the design choices made in this and subsequent chapters.

To keep principles of CCKit platform agnostic, the directory protocol is split into two components: a platform-agnostic *directory protocol state machine* and a platform-specific *implementation of the protocol state machine called DC* that is tailored to Enzian. The protocol state machine identifies the necessary state transitions on cache lines and actions to be performed to provide all the guarantees discussed above. Any deadlock scenarios in the protocol state machine is referred to as *protocol deadlocks*. This state machine should be derived from a formal specification of the directory protocol which would identify all possible coherence transactions that guarantee coherence invariants. Due to the highly concurrent nature of directory protocols, these state machine have hundreds of intermediate (or transient) states to achieve their goals and should be automatically generated from the specification.

The second component of the directory protocol is the actual implementation of the protocol state machine on the FPGA (i.e. DC). The protocol state machine does not take into account resource availability or limitations when providing its guarantees. It is the responsibility of the implementation to faithfully recreate the protocol state machine guarantees in spite of having limited resources. For example, the implementation should maintain a separate state machine for each cache line to ensure cache lines are mutually independent. In addition to the guarantees of the state machine, the implementation should also avoid deadlocks (due to resource conflicts) and be performant. These deadlocks due to resource conflicts are called *resource deadlocks*. Just because the protocol state machine is performant and does not have any *protocol deadlocks* does not mean its implementation is also performant and free of *resource deadlocks*. An example implementation of DC on Enzian is discussed in chapter 8.

In order to build the two components of directory protocol, we first need its specification. Although a list of coherence messages is available, the formal specification of the directory protocol (i.e. all possible coherence transactions that can occur) is not available to us due to the proprietary nature of the CPU's native protocol in Enzian. So by reverse engineering traces collected between two CPUs, we were able to identify certain rules of interaction between the coherence controllers (specifically between LLC and DC) and thus build a model that represents the directory protocol. Applying the rules of interaction to the protocol model allowed us to extrapolate all possible coherence transactions (i.e. the directory protocol specification) that would have to be handled by the protocol state machine. It is to be noted that *the protocol model and rules of interaction are not specific to Enzian and are applicable to any write-invalidate protocol with no NACKs (no-acknowledgments).* It is impossible to build a model that exactly represents the CPU's native protocol but having a model that is an approximation of reality, which can be tuned to new observations, is sufficient to build the protocol state machine.

Once we have all possible coherence transactions from the protocol model, we would have to specify them. In this chapter, we develop a notation for specifying coherence transactions called *state equations*. With this notation, we can specify the directory protocol and use it to automatically generate the protocol state machine. The idea is that *through proper design choices and by repeatedly applying certain operators, we will be able to solve these state equations to get a state machine that provides all guarantees required by the directory protocol*. An advantage of such a representation is any modifications to the coherence protocol can be incorporated in these state equations thereby making the protocol specification flexible.

In order to automatically generate the protocol state machine from its specification, we developed a *state space exploration tool*. This tool is introduced in this chapter and generates a deterministic state machine that shows all state transitions required by a *single* cache line for maintaining coherence invariants (subsection 2.3.4). Next, The tool checks for protocol deadlocks by looking for circular dependencies in the state transitions. Finally, it also explores the trade-off between state-space and performance to optimize for performance of the protocol state machine in the critical path. The need for customization is one reason to automate the generation of state machine. An application developer should be able to quickly change the protocol specification, generate a state machine and have it implemented without too many modifications to existing infrastructure.

The formalisms and techniques developed in this chapter are applicable to any symmetric coherent platform. Any examples would be based on ECI and Enzian. This chapter consists of the following sections.

1. Section 4.2 describes the model we have developed for directory protocol.
2. Section 4.3 describes the expected rules of interaction between CPU's LLC on one node and DC on the other node.

3. Sections 4.4 and 4.5 goes into details of coherence messages exchanged over a coherent interconnect as well as memory events.
4. Section 4.6 describes how we specify coherence transactions with *state equations* and the *operations* that are required to solve them. We also enumerate a number of design choices and the reason for making them.

Subsequent chapters will go into details of identifying all coherence transactions as well as how to build the state space exploration tool to generate the state machine.

4.2 Directory Protocol Model

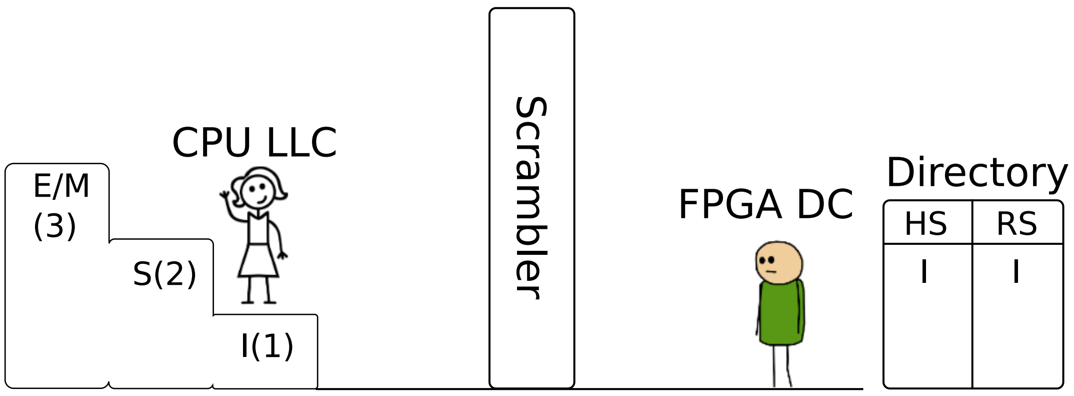


Figure 4.1: *Directory protocol model*

In this section, we model the interaction between the CPU and FPGA when the CPU coherently accesses the FPGA attached memory. We also rationalize the assumptions made by this model in representing a generic system, one that is not specific to Enzian.

The simplified model of the interaction between the LLC on the CPU and DC on the FPGA for a *single* FPGA-homed cache line is shown in Figure 4.1. Only considering a single cache line in the model ensures that all cache lines are treated identically by the model and no associations between cache lines are made (each cache line has a separate state machine whose transitions are mutually independent from transitions on other cache lines). On the left hand side is the CPU's LLC that can cache an *FPGA-homed* cache line in *I*, *S* or *Exclusive/Modified (E/M)* state. This is represented by the flight of stairs with each state enumerated as shown in figure (I is step 1, S is step 2 and E/M is step 3). On the right hand side is the DC that maintains a directory tracking the current state of the cache line in FPGA (Home State (HS)) and CPU (Remote State (RS)), based on messages received by it. Since there are no caches on the FPGA, the HS is always I. The RS is equivalent to which step the CPU is on as perceived by the DC. It is the responsibility of the DC to accurately track the remote state and maintain coherence invariants when handling requests from the CPU. The DC reads and writes to the FPGA memory through a separate *memory bus*, which is not shown here. For the remainder of this section, the terms *state* and *step* would be used interchangeably. Although this model is generic, its principles would be illustrated with examples from Enzian and ECI. A follow-up work that characterizes the coherence protocol on Enzian can be found at [Sch23].

Takeaway 4.1. ***Directory protocol guarantee: Transitions on cache lines are mutually independent.*** *We model the interaction between the CPU and DC for a single cache line. Only considering a single cache line in the model ensures that all cache lines are treated identically by the model and no associations between cache lines are made (each cache line has a separate state machine whose transitions are mutually independent from transitions on other cache line state machines).*

The CPU and DC communicate through a coherent interconnect (ECI in case of Enzian). We assume that the interconnect guarantees the delivery of in-flight messages but not its ordering. That is, if the CPU sends two messages one after the other, the DC *will* receive them but can receive them in either order. The guaranteed delivery of messages is a reasonable assumption as modern interconnects (including PCIe) have several layers that detect errors and request re-transmissions. The reordering of in-flight messages by the interconnect is modeled by the *scrambler* shown in Figure 4.1, which will be discussed in detail later in subsection 4.3.1. It is to be noted that the scrambler affects ordering of events only on the coherent interconnect and never on the memory bus.

We assume a typical MOESI protocol where an upgrade from E to M state in the CPU's LLC is silent (without any coherence message). As a result, the DC cannot distinguish between these two states, as shown by a single E/M step in Figure 4.1. Furthermore, there is no O state on the CPU. The CPU's LLC would get a cache line in O state only when a cache line that is dirty in FPGA's cache is transferred to the CPU's cache without being cleaned. Since there are no caches on the FPGA, this scenario would never occur.

Next, lets look at the rules of interaction imposed by the coherence protocol between these two coherence controllers.

Note 4.1. *In this chapter, the term CPU is used interchangeably with CPU's LLC and the term interconnect refers to the coherent interconnect.*

4.3 Rules of CPU-DC Interaction

In this section, we describe 7 rules of interaction between CPU's LLC and FPGA's DC in the context of the directory protocol model in Figure 4.1. Although we developed these rules by observing the communication traces between two ThunderX-1 CPUs, they are generic enough for any directory-based MOESI protocol *without* no-acknowledgments (NACKs). In addition, we also describe the properties that we expect of the underlying interconnect. The 7 rules of interaction are as follows.

Rule 1 (Going up the stairs)**.** *The CPU can make a request to go up the stairs and waits for a response from DC.*

- *The request can be from any lower step to any higher step (step 1 to 2, 1 to 3 or 2 to 3).*
- *The DC has to ensure coherence invariants and always reply with an acknowledgment (*No NACKs*). Upon replying the DC updates its directory.*
- *Once the CPU receives the response from DC it goes up to the requested step.*
- *This interaction is the* ***upgrade request-response pair*** *where the CPU wants to upgrade an FPGA-homed cache line in its LLC.*

Rule 2 (Coming down the stairs)**.** *The CPU can voluntarily come down the stairs anytime but cannot do so silently.*

- *The CPU can come down from any higher step to any lower step (step 3 to 2, 3 to 1 or 2 to 1) and it does so by issuing a message.*
- *The CPU does not require any response from the DC when coming down the stairs.*
- *The DC updates its directory when it receives the message.*
- *This interaction is a* ***voluntary-downgrade response*** *and occurs when the CPU wants to downgrade or evict an FPGA-homed cache line from its LLC. This interaction is* posted *i.e. does not require any response.*

Rule 3 (DC requesting CPU to go up the stairs)**.** *The DC cannot make the CPU go up the stairs.*

- *This means that an FPGA-homed cache line cannot be loaded by the DC into the LLC of the CPU without the CPU making an explicitly upgrade request.*

Rule 4 (DC requesting the CPU to come down the stairs)**.** *The DC can request the CPU to come down the stairs and wait for a response.*

- *The DC can request the CPU to come down from any step to any step.*
- *The CPU should send an acknowledge response and come down the stairs (no NACKs).*
- *Upon receiving the response, the DC updates its directory.*
- *This interaction is the* ***forward request-response pair*** *and occurs when the DC wants to either clean or clean-invalidate an FPGA-homed cache line that is present in the CPU's LLC.*

Rule 5 (No changing of minds)**.** *Once a message is issued by either the CPU or the DC for a cache line, the message cannot be canceled.*

- *For example, if the CPU requests to upgrade from I to S state, it cannot cancel this request to issue an upgrade from I to E/M state instead.*
- *This also means that for each coherence controller there can only be one on-going transaction for a cache line.*

Rule 6 (Properties of interconnect)**.** *The coherent interconnect is deadlock free, ensures delivery of messages but does not guarantee ordering. Delivery of messages is not instantaneous*

- *Guaranteed delivery means that both CPU and DC does not have to resend a message multiple times, and each message is received only once.*
- *Not guaranteed ordering means the messages can be received in any order by the counterpart.*
- *Not having immediate delivery implies that the interconnect has a latency in delivering messages.*

Rule 7 (Timeouts leading to system checks)**.** *Since there are no NACKs in the protocol, not receiving an expected coherence message implies an error. Coherence controllers can wait for a certain amount of time for reception of a coherence message before they time out and cause a system check. We assume that the timeout limit is much higher than rate at which coherence messages are exchanged. For example, in Enzian the CPU's timeout limit is in the order of milli-seconds whereas coherence messages are exchanged within hundreds of nano-seconds. There is no upper-limit on the timeout.*

Note 4.2. *Timeouts can arise only due to catastrophic failure such as complete failure of the interconnect or if the coherence protocol is not implemented correctly. In Enzian, the timeout mechanism is only implemented by the CPU's LLC (the current DC implementation has an infinite limit on the timeout). Timeouts in Enzian are almost always due to either the FPGA responding incorrectly or not responding at all.*

4.3.1 Rules of event reordering by the interconnect

Having seen the rules of interaction between the CPU and DC, we will look at the effect of reordering by interconnect next. The scrambler shown in Figure 4.1 models the fact that in-flight messages can be reordered by the interconnect and the reordering rule is as follows. If there are N in-flight messages sent by an entity, they can be received in $N!$ ways by its counterpart. For example, if there are three message M1, M2, M3 sent by the CPU to the DC in this order, they can be received by the DC in the following (3!) 6 ways.

- M1, M2, M3
- M1, M3, M2
- M2, M1, M3
- M2, M3, M1
- M3, M1, M2
- M3, M2, M1

The first thing of note is that, the scrambler can only affect the ordering of in-flight messages in the coherent interconnect, *it cannot affect the state of the cache line in the CPU's LLC or the end goal of a coherence transaction.* In other words, it cannot affect the step the CPU is in or the contents of the DCs' directory when a coherence transaction is complete. Next, the scrambler reorders only coherence messages and never events from the memory. Finally, by factoring all possible ways of reordering, this model is agnostic to the implementation specifics of interconnect. Let us next look at coherence messages exchanged between the CPU and DC through the coherent interconnect given the model and rules of interaction.

4.4 Coherence Messages

Applying the rules of interaction from section 4.3 to directory protocol model in Figure 4.1, lets us identify the coherence messages that can be exchanged between the CPU and the DC for an FPGA-homed cache line.

Note 4.3. *The coherence messages shown here are not specific to Enzian CPU's native protocol as they are obtained by applying a set of rules on a generic model, but there is a one-to-one mapping between them. This mapping is given in Table A.1.*

Upgrade request-response pairs: Rule 1 indicates that the CPU can request to go up from any lower step to any higher step and the DC has to acknowledge the request. Thus the CPU can make the following upgrade requests: request to go from step 1 to 2, request to go from step 1 to 3 or request to go from step 2 to 3. Furthermore the DC cannot deny (or NACK) the request and always has to acknowledge with a response. To describe a bit more on the naming convention for these messages, upgrade requests begin with the letter "R" followed by the lower step from which the CPU wants to upgrade from, and the higher step the CPU wants to upgrade to. For example, *R12* is request made by CPU to upgrade from step 1 to 2. Upgrade responses begin with "RA" (request acknowledge) followed by the higher step to which access was granted. For example, *RA2* is the response from the DC for an *R12* upgrade request allowing the CPU to go to step 2.

Since the CPU can request to go from any lower step to any higher step, we have the following upgrade request-response pairs.

- R12, RA2 (Request to upgrade from I to S, response has cache line data)
- R13, RA3 (Request to upgrade from I to E, response has cache line data)
- R23, RA3 (Request to upgrade from I to S, response *does not* have cache line data)

What this means outside the model is that for an FPGA-homed cache line, the LLC of the CPU can make upgrade requests from I to S state, or from I to E state or from S to E state to which the DC has to acknowledge. The upgrades from E to M state in the CPU's LLC are silent and the DC does not get notified. The responses for upgrade requests from I to S or E will carry cache line data (read from the FPGA memory) because the CPU is upgrading from I state and does not have any copies of the cache line. Whereas response to upgrade request from S to E does not carry any data as the CPU already has a copy of the cache line in S state.

Note 4.4. *An observation made in Enzian is that an upgrade to E by the CPU is always accompanied by a silent upgrade to M but we do not make this assumption when building the directory protocol.*

Voluntary downgrade response: Rule 2 says that the CPU can come voluntarily down the stairs from any higher to any lower step but has to send a voluntary downgrade response when doing so. These messages are do not require a response from the DC and are referred to as posted messages. In order to name these responses, we use the following convention: Voluntary downgrade responses start with the letter "V" followed by the higher step from which the CPU comes down, and then the lower step to which the CPU comes down to. For example, if the CPU comes down from step 2 to step 1, it issues a voluntary downgrade response *V21.* Since step 3 means the CPU can be either in E or M state, there are two flavors of coherence messages when the CPU comes down from step 3; One with dirty data and one without. The presence of dirty data is indicated by the letter "d". For example, if the CPU comes down from E to I (clean cache line), it issues a message *V31* whereas if CPU comes down from M to I with dirty data, the message issued is a *V31d.* Dirty data has to be written back to the FPGA memory since there are no caches on the FPGA. Note that when the CPU comes down from S to I, there cannot be any dirty data.

Thus we have the following voluntary downgrade response messages from the CPU.

- V21 (Voluntary downgrade from S to I)
- V31 (Voluntary downgrade from E to I, cache line is clean)
- V31d (Voluntary downgrade from E to I, cache line is dirty)
- V32 (Voluntary downgrade from E to S, cache line is clean)

- V32d (Voluntary downgrade from E to S, cache line is dirty)

In the real world, the voluntary downgrade responses are issued by the LLC of the CPU when downgrading (cleaning) or evicting (clean invalidating) a cache line from its cache. Downgrades of a dirty cache lines will be accompanied with data whereas downgrades of clean cache lines are not. The downgraded dirty data would have to be written back to the FPGA memory by the DC (since there are no caches on the FPGA).

Note 4.5. *Another observation made in Enzian is the CPU never voluntarily downgrades from E to S but we do not make this assumption when building the protocol.*

Forward downgrade request-response pairs: Rule 4 says the DC can request the CPU to come down the stairs. These requests from the DC are *forward downgrade requests* and are always accompanied by an acknowledgment *forward downgrade response* (the CPU cannot deny or NACK the request). Depending on what step the DC has the CPU in its directory, it can issue forward downgrade requests from step 3 to 2 or step 3 to 1 or step 2 to 1. These forward downgrade requests are named to begin with an “F” (forward) followed by the higher step, and then the lower step. For example, *F21* is the request issued by the DC when it wants the CPU to come down from step 2 to 1. The forward downgrade responses begin with an “A” followed by the step from which the CPU comes down from to the step the CPU goes down to. For example, the CPU would respond to *F21* with an *A21* and come down from step 2 to 1.

When it comes to step 3, there are no clean and dirty variants of forward *requests* as the DC does not have a way of knowing whether a cache line in E state is clean or dirty. But, just like in voluntary downgrade responses, there can be clean and dirty variants for the forward downgrade *response* from the CPU. For example, if the DC wants the CPU to come down from step 3 to 2, it issues *F32*, and the CPU can respond with *A32d* (dirty cache line) or with *A32* (clean cache line).

This gives the following forward downgrade request-response pairs.

- F21, A21 (Forward request, response to downgrade from S to I in CPU’s LLC)
- F32, A32 or A32d (Forward request, response to downgrade from E to S)
- F31, A31 or A31d (Forward request, response to downgrade from E to I)

These messages show that the DC can either *clean* or *clean-invalidate* an FPGA-homed cache line from the CPU's LLC. Cleaning a cache line means the cache line is either S or I state in the CPU's cache but never E or M. Clean-invalidating guarantees that the cache line is in the I state in CPU's cache. For both operations, any dirty data would be written back to the FPGA memory and, at this point, the FPGA memory has the only, most up-to-date copy of the cache line.

Takeaway 4.2. *The FPGA can clean or clean-invalidate an FPGA-homed cache line that is cached in CPU.*

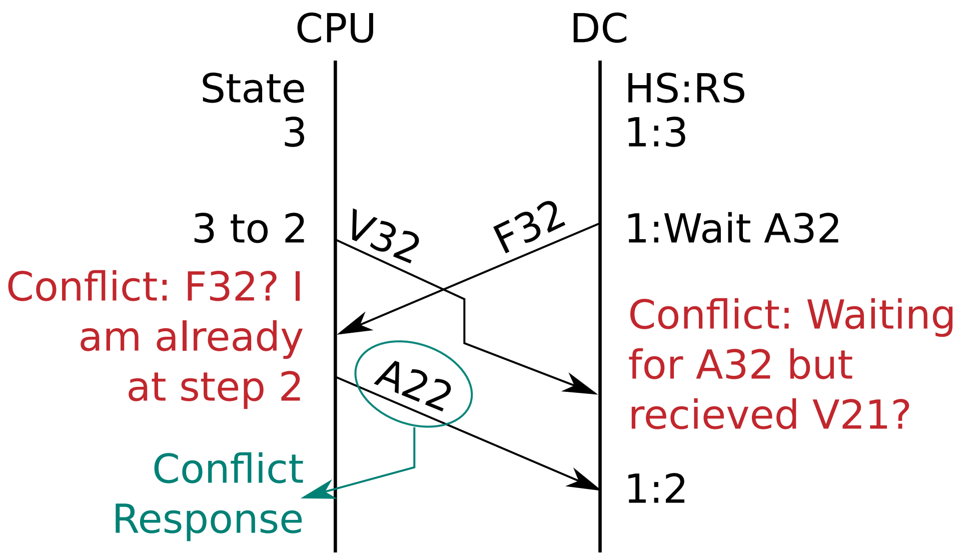

CPU
DC
State
3
HS:RS
1:3
3 to 2
V32
F32
1:Wait A32
Conflict: F32? I am already at step 2
Conflict: Waiting for A32 but recieved V21?
A22
Conflict Response
1:2

Figure 4.2: *Conflicts due to latency of interconnect: When a forward downgrade request from DC is in transit, the CPU voluntarily downgrades. These conflicts are resolved by exchanging additional messages called conflict responses.*

Forward downgrade conflict responses: Since the CPU can come down the stairs either voluntarily or at the request of the DC, conflicts arise when both operations are happening at the same time. For example, lets say the CPU is in step 3, which is also reflected in the directory of the DC. The DC issues an F32 requesting the CPU to come down from step 3 to 2. Since *the interconnect is not instantaneous*, this message takes sometime to reach the CPU. While F32 is in transit, the CPU decides to voluntarily come down from step 3 to step 2 by issuing a V32. From the CPU's perspective when it receives the F32, it is already at step 2, and from the DC's perspective, it is has received the voluntary downgrade response (V32) but is still waiting for a response for the forward downgrade request (F32). This race condition due to latency of the interconnect is shown in Figure 4.2 and to handle this, the CPU can issue conflict forward-downgrade responses in addition to the standard forward-downgrade responses. There are two forward-downgrade conflict responses namely *A22* and *A11*. An *A22* response from the CPU means that when the CPU received the forward-downgrade request to go to step 2, it was already at step 2 and that there might be voluntary downgrade responses in transit. Similarly, the CPU issues an *A11* when it receives any forward downgrade request (F32, F31, F21) when it is already at step 1, implying there is one or more voluntary downgrade responses in transit. Both these messages do not have a "dirty" variant as they are not issued when the cache line is in E state.

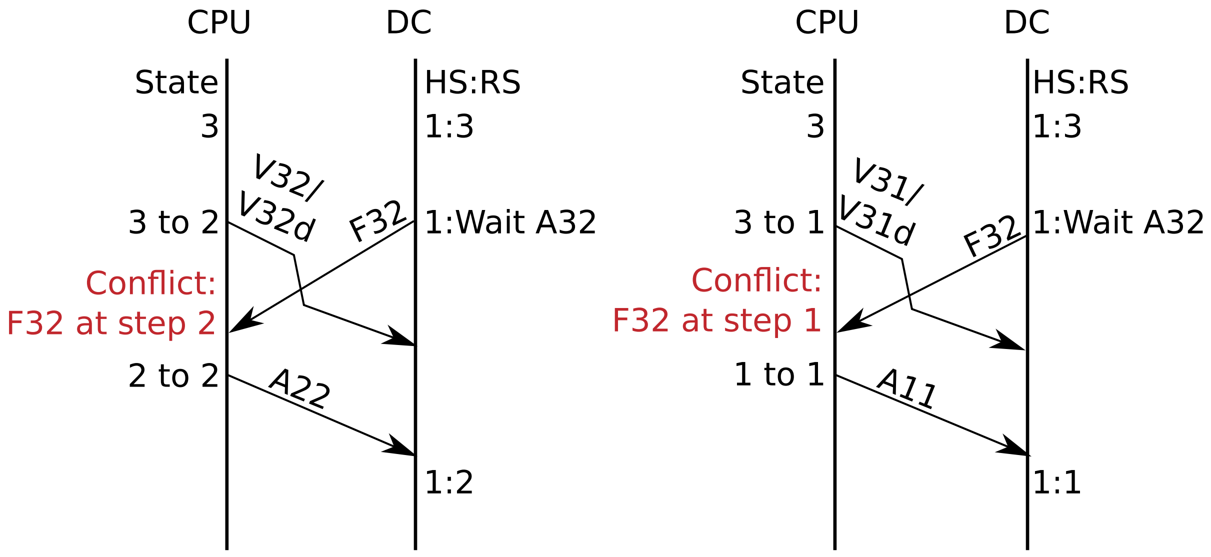


Figure 4.3: *Conflicts responses for F32: In left, DC has CPU in step 3 when issuing F32 but CPU is already at step 2 when receiving it. In right, DC has CPU in step 3 when issuing F31 but CPU is already at step 1 when receiving it. Conflict responses A22 and A11 are issued.*

Now lets enumerate all conflicts that can happen in our model due to latency of the interconnect. The first conflict scenarios are shown in Figure 4.3. In this scenario the DC has the CPU in step 3 and requests the CPU to come down from step 3 to 2 (issues F32). In this case, conflict arises when the CPU is in step 2 or step 1 when it receives the downgrade request. If the CPU is in step 2, it sends an A22 as a conflict response and remains in step 2. if the CPU is in step 1, it sends A11 as conflict response and remains in step 1. Note that if the CPU is already in step 1, the DC cannot make the CPU to go up the stairs to step 2, using the forward downgrade request F32.

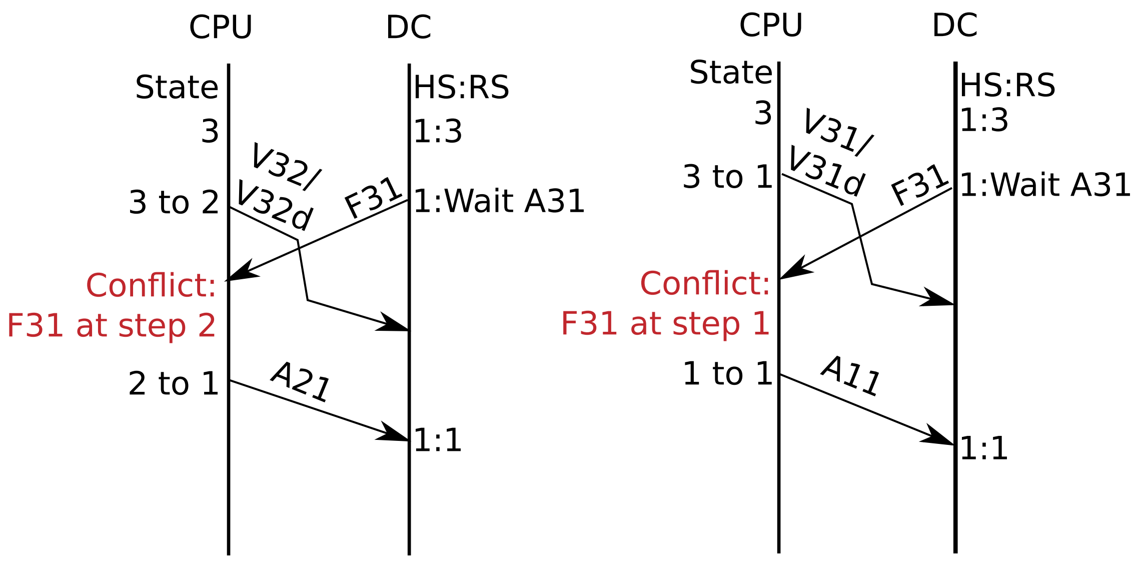


Figure 4.4: *Conflicts responses for F31: In left, DC has CPU in step 3 when issuing F31 but CPU is already at step 2 when receiving it. In right, DC has CPU in step 3 when issuing F31 but CPU is already at step 1 when receiving it. Conflict responses A21 and A11 are issued.*

In the second scenario (shown in Figure 4.4), the DC has the CPU in step 3 in its directory and asks the CPU to come down from step 3 to 1 by issuing an F31. Again conflict arises when the CPU is in step 2 or step 1 by the time it receives the downgrade request. If the CPU is in step 2, it has to come down to step 1 and it does so by issuing an A21 (acknowledge F31 but CPU coming down from step 2 to 1 instead of 3 to 1). If the CPU is already in step 1, it issues an A11 as conflict response to downgrade request and remains in step 1.

Finally in the third scenario, the DC has the CPU in step 2 and asks the CPU to come down from step 2 to 1 by issuing an F21. In this case conflict arises when the CPU is already in step 1 when it receives the downgrade request. The CPU resolves this conflict by issuing an A11 and remaining in step 1. This conflict scenario is shown in Figure 4.5

Takeaway 4.3. *Conflicts can arise due to the latency of interconnect given the highly concurrent nature of coherence protocols. These conflicts are handled by exchanging additional coherence messages. As will be seen later in subsection 5.5.3 another source of conflicts is the reordering of messages by the interconnect. There are no other sources of conflicts.*

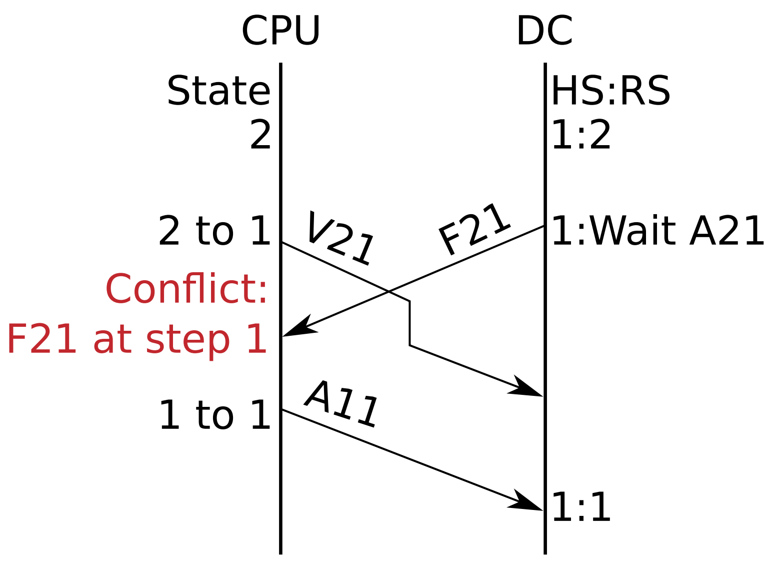


Figure 4.5: *Conflicts responses for F21: DC has CPU in step 2 when issuing F21 but CPU is already at step 1 when receiving it. Conflict response A11 is issued.*

Direction	Initiating Message	Expected Response	Conflict Response
CPU to DC	R12	RA2	-
	R13	RA3	-
	R23	RA3	-
CPU to DC	V21	-	-
	V31d/V31	-	-
	V32d/V32	-	-
DC to CPU	F21	A21	A11
	F31	A31d/A31	A21, A11
	F32	A32d/A32	A22, A11

Table 4.1: *Coherence messages involved in the DC protocol model.*

In this section, we have considered only coherence messages that are predicted by the model even if there are a lot more coherence messages (for example, atomic and coherent non-caching operations). Table 4.1 summarizes all messages in coherence transactions that are obtained by applying the rules of interaction to the directory protocol model. In the next section, we classify these coherence messages into specific message classes.

4.4.1 Coherence message classes

In section 4.4 we applied the CPU-DC rules of interaction to the directory protocol model to identify all the coherence messages that would be exchanged between the two entities. These coherence messages are exchanged through the coherent interconnect.

An interconnect typically classifies messages into a number of message classes and provides a dedicated VC for each message class to allow for deadlock free exchange of messages. The most basic distinction that an interconnect should make is between *requests and response message classes*. That is, the interconnect should provide separate VCs for request and response messages. To illustrate how deadlocks can occur due to a single VC, consider a controller that requires response *Rsp1* before it can handle request *Req1*. If the response is stuck behind the request in a VC as shown in Figure 4.6, the response would never be received by the controller which will lead to a deadlock. This problem is resolved by having separate VCs for request and response message classes.

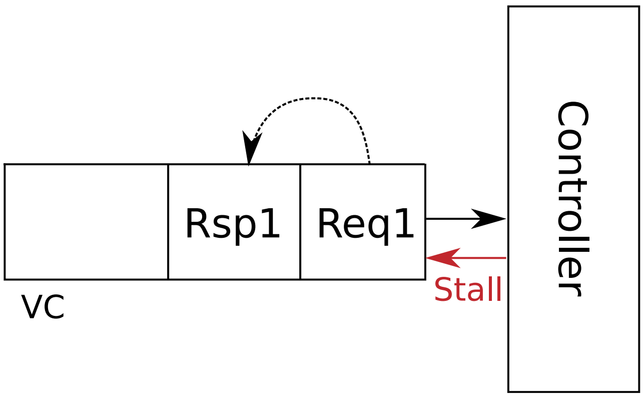


Figure 4.6: *Deadlock can arise when interconnect provides a single VC for both request and response message classes. Here the controller is waiting for a response to handle a request but the response is stuck behind the request in the VC. Having separate VCs for different message classes removes this problem.*

In addition to request-response classification, the interconnect can further classify messages based on the presence of data in the coherence message for performance reasons. Thus we can have separate VCs for *requests-without-data*, *requests-with-data*, *response-without-data* and *response-with-data*. Thus depending on the interconnect, the coherence messages shown in Table 4.1 would be classified into a number of message classes. We will now illustrate this classification on Enzian.

Section 3.3.1 describes the message classes that are provided by ECI. The classification of coherence messages in Table 4.1 into ECI message classes (and ECI VCs) is as follows.

Request-without-data message class: This message class comprises of upgrade requests (R12, R13, R23) that are made up of only a 64-bit header without any data. ECI assigns VCs 6 and 7 for this message class.

Response-with-data message class: This message class includes all response messages that have a header and up to 128-Bytes of cache line data. This includes voluntary downgrade responses with data (V31d, V32d), responses to upgrade requests with data (RA2, RA3) and responses to forward downgrade requests with data (A31d, A32d). ECI allows data transfer to happen at Sub-Cache-Line (SCL) granularity. A cache line can be split into 4 SCLs with each SCL being 32 Bytes. The number of SCL in a response-with-data message is identified using the *dmask* field in the header of the message. ECI assigns VCs 4 and 5 for this message class.

Response-without-data message class: This message class includes all response messages that only have a header and have no data associated. For example, the response RA3 that is sent for upgrade request R23 does not require data and this version of RA3 falls under response-without-data message class. Other messages in this class include voluntary downgrade responses without data (V31, V32, V21) and responses to forward-downgrade requests that do not carry data (A31, A32, A21, A22, A11). ECI assigns VCs 10 and 11 for this message class.

Forward-requests-without-data message class: The final message class includes forward-downgrade requests (F31, F32, F21) that have only a header and no data. ECI assigns VCs 8 and 9 for this message class.

Note 4.6. *ECI provides separate sets of VCs for odd and even cache lines indices to increase parallelism and improve performance. VCs with odd VC number carry events for* even *cache line indices and even VCs carry events for* odd *cache line indices. This is convention that is established by ECI and can vary with the interconnect.*

It is to be noted that there are more message classes associated with the DC (at its interface to ECI as well as local interface within the FPGA), I have only described a subset of them here and will describe additional message classes as needed. The message classes are summarized in Table 4.2

Takeaway 4.4. *Coherence messages are classified into a number of message classes. The implementation of the interconnect should provide separate VCs for each message class to*

ECI Message Class	VC #	Message
Request-without-data	6, 7	R12, R13, R23
Response-with-data	4, 5	V31d, V32d A31d, A32d
Response-without-data	10, 11	RA3 (for R23) V31, V32, V21 A31, A32, A21, A22, A11
Forward-request-without-data	8, 9	F31, F32, F21

Table 4.2: *ECI message classes and associated coherence messages.*

avoid deadlocks. The number of VCs provided per message class depends on the interconnect and its implementation.

4.5 Memory Transactions, Events and Message Classes

The DC provides coherent access to the FPGA attached memory. *Memory transactions* are initiated by the DC to read or write cache line data to FPGA memory. Memory transactions are classified into *read* and *write* transactions. The events associated with memory transactions are as follows:

Memory read request-response pair: A read transaction has a read request (RDD) event that is issued by the DC to memory followed by a read response event (RDDA) from the memory. Each event forms its own message class and have independent channels for communication.

Memory write request-response pair: A write transaction has a write request event (WDD) that is issued by the DC to memory followed by a write response event (WDDA) from the memory. Write events also form their own message classes and have separate channels for communication.

Although memory message classes have independent channels, they are not assigned a VC number. The memory message classes are summarized in Table 4.3. It is to be noted that memory events and transactions are agnostic to the interconnect and its coherence protocol.

Message Class	VC #	Message
Read-request	NA	RDD
Read-response (with data)	NA	RDDA
Write-request (with data)	NA	WDD
Write-response	NA	WDDA

Table 4.3: *Memory message classes and associated memory events.*

4.6 Specifying Coherence Transactions

As defined in chapter 2 coherence transactions are a chain of coherence messages exchanged between the DC and the CPU towards a common goal. These coherence transactions form the specification and are required to generate the protocol state machine.

Since the model described in section 4.2 sets the rules of communication between CPU's LLC and FPGA's DC, it is also used to identify all coherence transactions that are to be handled by the DC. Before delving into identifying all coherence transactions, let look at how they are specified using *state equations*.

4.6.1 State equation representation of coherence transactions

We have developed a notation called *state equations* to specify coherence transactions. We use state equations to specify coherence transactions from the *perspective of DC: The state maintained by it and the coherence events it sends and receives.*

The DC maintains both home and remote state of a cache line in its directory. The DC can then issue or receive coherence messages as part of a coherence transaction that can alter these states and require the DC to perform certain actions to maintain coherence invariants. Eventually the coherence transaction completes leaving the cache line in the state intended by the coherence transaction. These coherence transactions are described using state equations that have the following format.

$$Initial\ HS : RS,\ \{M1, M2, M3...\}\ \rightarrow\ Final\ HS : RS,\ (Action) \tag{4.1}$$

Where $Initial\ HS : RS$ is the initial home and remote states of a cache line in DC's directory, $\{M1, M2, M3...\}$ is the *ordered set* of coherence events (both coherence messages and memory events) *received* by the DC, *Final HS* : *RS* is the home and remote state of the cache line after all coherence messages are handled. (*Action*) is the action to be performed by the DC such as issuing coherence responses and initiating coherence and memory transactions.

A coherence transaction is always associated with *a single coherence request (or none at all).* As such, the state equation that represents a single coherence transaction can have *atmost one coherence request* in its ordered set of coherence events and not more.

Each coherence event that is received by the DC can be associated with an action that has to be performed by the DC. When the action is performed is determined by the specification of coherence transactions and the constraints imposed on DC protocol state machine in order to maintains coherence invariants. As will be seen in subsequent sections, actions for certain coherence events will be performed as soon as the coherence event is received and for other coherence events, the state of the cache line would determine when the action is performed.

Design Choice 4.1. *We use a state equation to specify a coherence transaction. The state equation represents the coherence transaction from the DC's perspective. This choice helps us to record the state transitions and the actions that should be performed by the DC within the state equation. The alternate way of specifying transactions from CPU's perspective would be useful for generating test cases.*

4.6.2 Specification of coherence transactions

A specification of the coherence protocol contains only the *subset of state equations in which the ordering of events is linearizable.* There is an inherent ordering of coherence events which gets reordered due to the nature of the system. For example, the CPU issues coherence events for a cache line in a specific order which can be reordered by the interconnect. Furthermore, coherence events can initiate memory transactions in which a memory request is always followed by a response. The specification describes all coherence transactions in the protocol where the ordering of coherence messages is the same as the order in which they were issued by the CPU. In the specification, memory operations are initiated immediately following its coherence event and memory operations are atomic, i.e. that the response is received as soon as the request is sent.

Note 4.7. *Linearizablity is ordering within a transaction. For example, a coherence event that initiates a memory request is always followed by a memory response. In contrast,* *serializability is across transactions. For example, memory transactions are serialized when the second transaction starts after the first transaction completes. Serialization of transactions leads to performance bottlenecks.*

Since we did not have a formal specification of the CPU's coherence protocol, we base the specification on the DC protocol model. Using this model, we idetnfy all pathways the CPU can take given an initial state and use this to identify the order of coherence messages issued by the CPU. The model also provides a layer of abstraction from the actual implementation of the coherence protocol and thus by changing just the model and applying the same principles we can build the protocol state machine of other implementations like CXL.

Given a state equation from the specification, it is the responsibility of the *state space exploration tool* to identify all possible reorderings that can happen and build the state machine. In order to do this, we need to understand how the scrambler in the protocol model affects a state equation.

Takeaway 4.5. *A specification of directory protocol contains only the set of state equations where the ordering of coherence events is the same as the order in which the events were issued by the CPU. It is the responsibility of the state space exploration tool to identify state equations that arise due to reordering when building the state machine.*

4.6.3 Effect of scrambler on state equations

Section 4.3.1 describes how the *scrambler* models the reordering of coherence messages by the interconnect in the DC protocol model. Since the scrambler cannot affect the state of the cache line in the CPU's LLC or the intended goal of a coherence transaction, it follows that the scrambler cannot affect the initial or final state of the cache line in the DC's directory. What it does affect is the number of coherence transactions that would have to be handled by the DC, that is, the number of DC state equations.

With this in mind, consider a state equation in which the CPU issues three messages M1, M2, M3 in that order. The DC can receive the message in 3! ways giving us 6 possible coherence transactions (with state equations shown in equation 4.2) to be handled by the DC for the same initial and final DC states and action to be performed.

$$\begin{aligned}
&Initial\ HS : RS,\ \{\boldsymbol{M1, M2, M3}\} \rightarrow Final\ HS : RS,\ (Action) && \text{specification}\\
&Initial\ HS : RS,\ \{\boldsymbol{M1, M3, M2}\} \rightarrow Final\ HS : RS,\ (Action) && \text{reordered}\\
&Initial\ HS : RS,\ \{\boldsymbol{M2, M1, M3}\} \rightarrow Final\ HS : RS,\ (Action) && \text{reordered}\\
&Initial\ HS : RS,\ \{\boldsymbol{M2, M3, M1}\} \rightarrow Final\ HS : RS,\ (Action) && \text{reordered}\\
&Initial\ HS : RS,\ \{\boldsymbol{M3, M1, M2}\} \rightarrow Final\ HS : RS,\ (Action) && \text{reordered}\\
&Initial\ HS : RS,\ \{\boldsymbol{M3, M2, M1}\} \rightarrow Final\ HS : RS,\ (Action) && \text{reordered}
\end{aligned} \tag{4.2}$$

We also noted previously that only coherence messages sent through the interconnect can be reordered by the scrambler and memory responses received by the DC cannot be reordered. For example, consider the following state equation in 4.3 where the CPU issues coherence message M1 followed by another coherence message M2 and message M1 causes the DC to issue a write request. Since memory transactions are represented atomically in the specification, the coherence message M1 will be followed by a write response (WDDA) and then coherence message M2. Even if there are three coherence events (2 coherence messages and 1 memory response event) there can be only 2! reordering since a write response cannot precede the coherence message M1 which issues the write request.

$$\begin{aligned}
Initial\ HS : RS,\ \{M1, WDDA, M2\} &\rightarrow Final\ HS : RS,\ (Action) && \text{specification}\\
Initial\ HS : RS,\ \{M2, M1, WDDA\} &\rightarrow Final\ HS : RS,\ (Action) && \text{reordered}\\
Initial\ HS : RS,\ \{\textbf{WDDA,M1}, M2\} &\rightarrow Final\ HS : RS,\ (Action) && \text{not possible}
\end{aligned} \tag{4.3}$$

Takeaway 4.6. *With the specification of a coherence transaction in the form of state equation without any reordering, it is possible for the state space exploration tool to identify all possible reordering scenarios and generate corresponding state equations.*

Messages in transit: It was noted earlier in subsection 4.6.1 that the state equation can have at most one coherence message of the request message class. Thus, in a state equation with multiple coherence events in its ordered set, any messages that are present after a request message indicate responses messages that are in transit and would be eventually received by the DC.

4.6.4 DC protocol state machine design choices

Before enumerating all possible state equations predicted by the protocol model and generating the protocol state machine, let us look at some design choices made for building the protocol state machine. The main aim here is to define the scope of the state space exploration tool that is used to generate the protocol state machine.

In the previous section we have seen that the DC state equations are specified at the granularity of a coherence transaction; it gives the final state and action given an initial state and *set* of coherence events. The protocol state machine that handles these transactions have to account for the fact that not all events in the coherence transaction arrive simultaneously. As such the state machine would benefit from a specification at the granularity of a single coherence event rather than with the set of coherence events in a transaction.

For example, consider a coherence transaction with two coherence events (M1, M2) out of which only the first event M1 has arrived at a given point.

$$Initial\ HS : RS,\ \{\mathbf{M1}, M2\}\ \rightarrow\ Final\ HS : RS,\ (Action) \tag{4.4}$$

Instead of the protocol state machine waiting for the second coherence event (M2) to arrive before handling the coherence transaction, it can consume the first event into an *intermediate* state and continue handling other coherence transactions. Eventually when the second event arrives, the state machine can complete the coherence transaction using information from the intermediate state. This effectively splits the coherence state equation in 4.4 into two sub equations as shown below.

$$\begin{gathered} Initial\ HS : RS,\ \{M1\}\ \rightarrow\ \textbf{Intermediate HS:RS} \\ \textbf{Intermediate HS:RS},\ \{M2\}\ \rightarrow\ Final\ HS : RS,\ (Action) \end{gathered} \tag{4.5}$$

It is the responsibility of the *state space exploration tool* to split the state equations into smaller equations and identify the intermediate states required. Having intermediate states increases the number of states and thus the complexity of the protocol but it improves the performance of the state machine by allowing it to handle multiple outstanding coherence transactions on different cache lines (i.e. avoiding serialization of transactions).

If the state space exploration tool has information on message classes and their VCs (which is specific to an interconnect), it can further optimize the state machine by looking to consume more than one event at a given time. For example, if there is a coherence transaction with three coherence events out of which two have arrived at a given point in different VCs, it can consume both events simultaneously by transitioning to an intermediate state as shown in equation 4.6. Although this optimization can improve performance in this very specific (non-critical) scenario, it increases the number of states and the complexity of the state space exploration tool.

$$\begin{aligned} \mathit{Initial\ HS} : \mathit{RS},\ \{\mathbf{M1,M2}\} &\rightarrow \mathit{Intermediate\ HS} : \mathit{RS} \\ \mathit{Intermediate\ HS} : \mathit{RS},\ \{\mathbf{M3}\} &\rightarrow \mathit{Final\ HS} : \mathit{RS},\ (\mathit{Action}) \end{aligned} \tag{4.6}$$

In order to keep the state space exploration tool simple, we choose to *not* perform this optimization. At the cost of being able to consume only one coherence event at a time, we limit the number of states and keep the state space exploration tool agnostic to message parallelism provided by VC (and thereby agnostic to the nature of the interconnect).

Design Choice 4.2. *The protocol state machine handles only one coherence event at a time even if there are multiple outstanding coherence events belonging to a coherence transaction on different VCs. This design choice makes the state machine generation process agnostic to the implementation details of the interconnect but removes the possibility of performing certain performance optimizations.*

Cardinality of state equations: The number of events in the ordered set of events in a state equation is the *cardinality* of a state equation. For example, all state equations in equation 4.2 have three coherence events and thus have a cardinality of 3. When comparing two state equations, the *smaller* state equation has lesser cardinality than *larger* state equation. The cardinality of a state equation is always a positive integer greater than or equal to 1.

Reducing state equations: The process of converting a state equation of cardinality $N(\forall N > 1)$ to multiple state equations of cardinality $K(\forall K < N)$ with intermediate states is called *reducing* the state equation.

Solving state equations: As per our design-choice 4.2, it is beneficial to reduce state equations with cardinality $N > 1$ to state equations with cardinality of 1 to build the state machine. The process of reducing a state equation of cardinality $N(\forall N > 1)$ to multiple state equations of cardinality 1 is called *solving* state equations.

Deterministic state machine: The second design choice made is that we want the state space exploration tool to generate a deterministic state machine where the next state and action to be performed is only determined by the present state and coherence event being handled. This helps simplifying the state machine to a simple table where the present state and coherence event can be looked up to get the next state and action to be performed.

Design Choice 4.3. *The protocol state machine has to be deterministic where the next state and action to be performed only depend on the present state and coherence event being handled. This is done to reduce the state space by collapsing equivalent states into a single state.*

4.7 State Space Exploration Tool

Putting it all together, we have seen the DC protocol model and its rules of interaction. Using these we identified all coherence messages that can be exchanged and their message classes. We have developed a notation to specify coherence transactions, although we havent specified any. In the subsequent chapters we will apply the rules of interaction to the protocol model to identify the coherence transactions in the directory protocol specification. We also made certain design choices that will help simplify the state space exploration tool.

The state space exploration tool is a term rewriting system [BKdV03] which repeatedly applies certain operators on state equations to reduce them. This tool takes in a specification of state equations that represent coherence transactions with a linearized order of events. The tool identifies all possible reordering of the state equations and reduces them to state equations of cardinality 1 (solves them) with intermediate states. These single-event state equations are then used by the tool to generate a deterministic DC protocol state machine that handles one coherence event at a time. The generated state machine should be performant, deadlock free and guarantee that coherence invariants are met. This state machine is represented in the form of a table Table 4.4 where given a present state and a coherence event, each cell identifies the next state and action to be performed. Next lets look at what operators can be applied by the tool on state equations to reduce them.

4.7.1 Operators on state equations

The state equation or cardinality 1 has only one coherence event and the state transition is directly specified in the state equation. For example equation 4.7 shows two state equations

Present HS:RS	M1	M2
S1	S2 (Action)	S3 (Action)
S2	S4 (Action)	S5 (Action)

Table 4.4: *State machine in the form of a table where each cell identifies the next state and action to be performed given a present state and coherence event. S1 to S5 are states and M1, M2 are coherence events.*

with a single coherence event where given the initial state S1 the state transitions are defined when coherence event M1 or M2 is received.

$$\begin{aligned} S1,\ \{M1\} &\rightarrow S2,\ (Action1) \\ S1,\ \{M2\} &\rightarrow S3,\ (Action2) \end{aligned} \tag{4.7}$$

These state equations can directly be represented in the state machine as shown in Table 4.5 where given a state, one can look up the next states and actions for different coherence messages.

Present HS:RS	M1	M2
S1	S2 (Action1)	S3 (Action2)

Table 4.5: *State machine for state equations with a single coherence event can be trivially generated.*

Slightly more complex are state equations with two coherence events (cardinality of state equation is 2). Since the state machine handles only one coherence event at a time (design-choice 4.2), these state equations have to be split into smaller state equations. Equation 4.8 gives an example where a state equation with two coherence events is split into two state equations with one coherence event each. This is done with S2 as a new intermediate state.

$$\begin{aligned} S1,\ \{M1, M2\} &\rightarrow S3,\ (Action1) \quad \text{given} \\ S1,\ \{M1\} &\rightarrow S2,\ (Action2) \quad \text{split 1} \\ S2,\ \{M2\} &\rightarrow S3,\ (Action1) \quad \text{split 2} \end{aligned} \tag{4.8}$$

The split equations can then be represented in the form of a table shown in Table 4.6 where X represents state transitions that are not possible given these state equations.

Thus to generate the state machine, multi-event state equations have to be reduced to single-event state equations by identifying intermediate states. To do this, we define

Present HS:RS	M1	M2
S1	S2 (Action2)	X
S2	X	S3 (Action1)

Table 4.6: *State equations with more than one coherence event has to be split down to state equations with one coherence event to be able to generate the state machine.*

three operators on state equations: **substitution** and **creation** and **stall operator for reordering**. The substitute and create operators help split state equations with two or more coherence events into state equations with lesser number of coherence events. When these operators are applied repeatedly, we can reduce the cardinality of state equation to 1. Thus the substitution and creation operators are together called *reduction* operators. The smaller state equation that is obtained by applying the reduction operators on a larger state equation is referred to as the *inferred* state equation.

In contrast, the *stall* operator does not split larger state equations into smaller state equations but rather reorders a state equation. Let us first look at the substitution operator.

4.7.1.1 Substitution operator

We can define a substitution operator where a *known* state equation with cardinality 1 is substituted into a given (specified) state equation with larger cardinalty N to create a new state equation with smaller cardinality $K(\forall K < N)$. This is akin to the state machine handling the first event in the ordered set of events in the given state equation and updating the directory with the new state prescribed by the known state equation. Substitution results in a new state equation whose present state is dictated by the known state equation and the ordered set of events is the same as the set of events in the given state equation but with the first event removed (as it has been already handled). Shown below in Equation 4.9 is an example of substituting a known state equation into a given state equation with two coherence events.

$$\begin{aligned} S1, \{\mathbf{M1}, M2\} \rightarrow S3, (Action1) &\quad \text{Given} \\ S1, \{M1\} \rightarrow S2, (Action2) &\quad \text{Known} \\ S2, \{M2\} \rightarrow S3, (Action1) &\quad \text{Inferred by substituting known in given} \end{aligned} \tag{4.9}$$

To explain, the given (or specified) state equation begins with initial state S1, receives two coherence messages M1 and M2 before transitioning to state S3. If, for example, the specification provides a smaller state transition from state S1 to S2 when encountering coherence event M1, then this can be *substituted* into the given state equation by replacing the present state (S1) and first coherence event (M1) with the state described by the known state equation (S2). Equation 4.9 also shows the resulting state equation with one less coherence event (indicating that the first event has been handled by the state machine). New state transitions can be *inferred* from the resulting state equation that has to be honored by the state machine. In the example above, it can be inferred from the resulting state equation that when handling coherence event M2 given present state S2, the next state should be S3 and *Action1* has to be performed. It can also be inferred that *Action2* has been completed.

This gives us an idea of “knowing” state equations. A state equation in the specification with a single coherence event is a “known” state equation by default. Any *single-event* state equation that are inferred will also be added to the compendium of known equations. Thus known state equations are single-event state equations that are either specified or inferred and known state equations can be substituted into larger state equations to get smaller state equations.

Algorithm 1 shows the pseudo code of the substitution operator. As shown the substitution operator replaces the current state and first event in the list of events with a known intermediate state to infer a new state equation.

Since the inferred state equation prescribes what the state machine has to do, it has to be *consistent* with the prescriptions of previously known state equations if the state machine has to be deterministic. For example in Equation 4.10, say we already know that the state machine transitions from S2 to S4 when event M2 is encountered but a newly inferred state equation prescribes a transition from S2 to S3 when M2 is encountered, then the inferred state equation is *inconsistent* with a previously known state equation and leads to non-determinism. Such inconsistencies point to a problem with the specification of coherence transactions.

$$\begin{array}{ll} S2, \{M2\} \rightarrow \textbf{S4, (Action4)} & \text{Known} \\ S2, \{M2\} \rightarrow \cancel{S3, (Action1)} & \text{Result inconsistent with known} \end{array} \tag{4.10}$$

The state space exploration tool can check for any inconsistencies when an inferred state equation is being added to the set of known state equations. Algorithm 2 shows a pseudo code of how this can be done. It checks an inferred state equation with a set of known

Algorithm 1 Substitution operator to reduce state equations: substitute known smaller equation into larger equation to reduce the larger equation and infer new state equations.

```
    equation $S1, \{M1, M2\} \rightarrow S3, (Action1)$
    Known $S1, \{M1\} \rightarrow S2, (Action2)$
 1: cur_stt ← S1
 2: evt_lst ← [M1,M2]
 3: fnl_stt ← S3
 4: acn ← Action1
 5: known[(S1, M1)]["next_state"] ← S2
 6: known[(S1, M1)]["action"] ← Action2
 7: procedure SUBSTITUTION($known, cur\_stt, evt\_lst, fnl\_stt, acn$)
 8:     int_stt ← $cur\_stt$
 9:     if (cur_stt,evt_lst[0]) in known then                ▷ Substitute
10:         cur_evt ← evt_list.popleft()
11:         int_stt ← known[(cur_stt, cur_evt)]["next_state"]
12:     cur_stt ← int_stt
13:     return (cur_stt, evt_lst, fnl_stt, acn)
14: Output: Reduced state equation
15: cur_stt ← S2
16: evt_lst ← [M2]
17: fnl_stt ← S3
18: acn ← Action1
```

state equations. If the state transition or action prescribed by the inferred state equation differs from those of known state equations then the inferred state equation is inconsistent.

Thus substitution operator when applicable, splits a larger state equation into a set of smaller state equations which can easily be converted into a state machine.

Takeaway 4.7. ***known*** *state equations are single-event state equations that are either specified and inferred. Known state equations can be substituted into larger state equations to get smaller state equations that would have to be honored by the state machine. New state transitions can be* ***inferred*** *from the resulting state equation and this should be* ***consistent*** *with previously known state equations. Any inconsistencies between previously known and newly inferred state equations will lead to non-determinism, indicating a problem with the specification of the coherence protocol.*

Algorithm 2 State equation consistency check: Check if an inferred state equation is consistent with previously known state equations.

```
1: procedure CHECK_SE_CONSISTENCY($known, cur\_stt, cur\_evt, fnl\_stt, acn$)
2:     if (cur_stt,cur_evt) in known then
3:         if fnl_stt $\neq$ known[(cur_stt,cur_evt)]]["next_state"] or
            acn $\neq$ known[(cur_stt,cur_evt)]]["action"] then
4:             return ("inconsistent")
5:     return ("consistent")
```

4.7.1.2 Stall operator for reordering state equations

Stall operators are intended to reorder the set of coherence events in a state equation. When conflicts arise due to interconnect reordering coherence events, a stall operator can be useful to identify the correct order of events as issued by the CPU. In contrast to substitution and creation operators, a stall operator can *never* create a new state equation. From the state machine point of view, a stall operator on the state equation is similar to the state machine delaying a coherence event from being handled till another coherence event is received. Delaying a coherence event does not require transitioning to an intermediate state.

$$\begin{aligned} S1, \{\mathbf{M1,M2}\} &\rightarrow S3, (\mathit{Action}1) && \text{Ordering of events by CPU} \\ S1, \{\mathbf{M2,M1}\} &\rightarrow S3, (\mathit{Action}1) && \text{Effect of reordering by interconnect} \\ S1, \{M2\} &\rightarrow S1, (\mathit{Stall}) && \text{Stalling M2 till M1 arrives} \\ S1, \{\mathbf{M1,M2}\} &\rightarrow S3, (\mathit{Action}1) && \text{Stalling results in reordering of events} \end{aligned} \tag{4.11}$$

It was noted earlier in subsection 4.6.1 that a state equation can contain at most one message of the request message class. It was also noted that any messages in the ordered message set of a state equation, that is present after a request message indicate response messages that are in transit and would be eventually received by the DC. In this scenario, stalling the request message till one or more messages in transit are received can be beneficial. For example, stalling an upgrade request till all downgrade responses with dirty data is received is essential to maintain the *data-value* invariant. That said, stalling coherence events can lead to deadlocks and requires careful design choices to avoid them. This is discussed in detail in section 5.5.

4.7.1.3 Create operator to create intermediate state

Algorithm 3 Create new intermediate state: Whenever a state transition is not known, create an intermediate state in the format given by the algorithm.

```
    Input $S1, \{M1, M2, M3\} \rightarrow S3, (Action1)$
    Output int_stt for $S1, \{M1\} : S3\_M2\_M3$
 1: cur_stt ← S1
 2: evt_lst ← [M1,M2,M3]
 3: fnl_stt ← S3
 4: acn ← Action1
 5: cur_evt ← evt_lst.popleft()
 6: procedure CREATE_INT_STT($known, cur\_stt, evt\_lst, fnl\_stt, acn$)
 7:     int_stt ← fnl_stt
 8:     N ← length(evt_lst)
 9:     for $k \leftarrow 0$ to $N - 1$ do
10:         int_stt ← append(int_stt, ‘‘_’’, evt_lst[k])
11:     return (int_stt)
```

A create operator can be used to reduce state equations when a substitution is not possible. The create operator creates a new *intermediate* state which can then be substituted in the larger state equation to get a smaller state equation.

For example, consider the following state equation (equation 4.12). Here the state of the cache line is initially S1 when coherence event M1 is received. The specification does not indicate the next state and action for this situation. Assuming maintaining data-invariants allows for action corresponding to M1 be performed (M1 is not stalled), we can consume the message M1 into an intermediate state *IS1* and perform action corresponding to the message M1. This transition is deterministic and should be added to the set of known equations to avoid non-determinism.

$$\begin{aligned}
S1, \{\mathbf{M1}, M2, M3\} &\rightarrow S3, (Action1) && \text{given} \\
S1, \{M1\} &\rightarrow unknown, (Action\ for\ M1) && \text{if M1 is not stalled} \\
S1, \{M1\} &\rightarrow \mathbf{IS1}, (Action\ for\ M1) && \text{create new state} \\
& && \text{add to known} \\
IS1, \{M2, M3\} &\rightarrow S3, (Action1) && \text{inferred by substitution}
\end{aligned} \tag{4.12}$$

Substituting the intermediate state into the original state equation will result in a smaller state equation that would have to be honored by the DC protocol.

$$\begin{aligned}
S1, \{M1, M2, M3\} &\rightarrow S3, (Action1) && \text{given} \\
S1, \{M1\} &\rightarrow \mathbf{S3\_M2\_M3}, (Action\ for\ M1) && \text{create new state} \\
& && \text{add to known} \\
\mathbf{S3\_M2\_M3}, \{M2, M3\} &\rightarrow S3, (Action1) && \text{inferred by substitution}
\end{aligned} \tag{4.13}$$

The name of the intermediate state can be arbitrary and the convention shown in equation 4.13 is used for naming intermediate states. The intermediate state name contains the final state and remaining coherence messages from the given state equation, separated by underscore. This is done to improve readability. The pseudo code is given in algorithm 3.

4.8 Summary

In pursuit of generating the DC protocol state machine, so far we have developed a model for the directory protocol and identified all coherence events that would have to be involved in the model. We also looked at how coherence transactions can be specified using state equations and how solving them can help generate the protocol state machine. We then looked at a few basic operations that can be used by the state space exploration tool to solve state equations though we havent seen how. Finally we have seen the first set of design choices made before building the DC protocol state machine.

The main reason to build a model, specify the protocol and have the protocol state machine generated automatically is to keep everything customizable and agnostic to the nature of the interconnect.

As for the guarantees that are to be provided by the DC protocol state machine, we have only ensured one: Coherence transactions on different cache lines would be independent of each other. We have also seen how conflicts arising due to latency of interconnect can be handled by issuing a special class of conflict responses. In the next chapter we look at how to enumerate coherence transactions from the model and build the first version of the DC protocol state machine.

5 Specifying Coherence Transactions Initiated by CPU

5.1 Introduction

In chapter 4 we introduced the DC protocol model along with the rules of interaction between the CPU and the DC on the FPGA for accessing an FPGA-homed cache line. We also looked at the notation of state equations and how they can be used to specify coherence transactions. The next question is *how do we identify all possible coherence transactions that can occur in our model?* To answer this question, we look at a subset of coherence transactions in this chapter; transactions that are initiated by the CPU. We specifically answer the question: *how do we identify and specify all possible coherence transactions that can be initiated by the CPU?*

The second question this chapter begins to answer is *what would be the algorithm of the state space exploration tool?* As mentioned earlier, the state space exploration tool must generate the directory protocol state machine from its specification. Based on the guarantees that are to be provided by the directory layer (discussed in section 3.4), we define the goals of this tool as follows. First, it has to account for the fact that the interconnect can reorder coherence messages. Second, it has to guarantee the coherence invariants (discussed in subsection 2.3.4) are met. Third, it should guarantee there are no deadlocks in the DC protocol state machine. Finally, it should optimize the state machine for performance in its critical path. This chapter addresses how such a state space exploration tool can be built.

Since we do not have a formal specification of coherence transactions (subsection 4.6.2 for more details), we rely on the DC protocol model to create our own specification. To begin with, we identify certain initial conditions and then apply the rules of interaction to get all coherence transactions that are possible given the initial conditions.

We limit the number of initial conditions by only considering coherence transactions that are initiated by the CPU in this chapter. This means that forward-downgrade transactions that can be initiated by the DC are not considered. In other words, *we only consider transactions that allow the CPU to coherently access the FPGA attached memory.* In subsequent chapters we will relax this constraint to allow forward-downgrade transactions and application issued coherence messages.

Thus we have three main objectives in this chapter. The first is to use the DC protocol model from section 4.2 to obtain a specification of the DC protocol. The second objective is to develop an algorithm for the state space exploration tool. The final objective is to construct a DC protocol state machine using the state space exploration algorithm, that allows CPU to coherently access FPGA address space.

This chapter does not make any new assumptions on the nature of the interconnect and is organized as follows.

1. Sections 5.3 and 5.2 establishes three initial conditions in the DC protocol model from which we can build the DC protocol state machine.

2. Sections 5.4, 5.5 and 5.6 goes into detail of building the protocol state machine given the initial conditions. All these sections are structured to address how the goals of state space exploration tool are achieved.

    - A section to identify all pathways the CPU can take.
    - A section to build the specification state equations based on these pathways and how identify how coherence invariants can be maintained for these coherence transactions.
    - A section to consider reordering of coherence transactions and identify how coherence invariants can be maintained for each coherence transaction.

- A section to build the actual state machine that is deadlock free and performant.

3. Section 5.6 distills the method of solving state equations into an algorithm which can be used to automatically generate the state machine.

Takeaway 5.1. *Goals of state space exploration tool: First, it has to account for the fact that the interconnect can reorder coherence messages. Second, it has to guarantee the coherence invariants are met. Third, it should guarantee there are no deadlocks in the DC protocol state machine and finally, it should optimize the state machine for performance in the critical path.*

5.2 Initial Conditions

An initial condition is where state of a cache line is one of the stable states (I, S or E) and the state of the cache line in the CPU's LLC matches the remote state (RS of HS:RS shown in Figure 4.1) of the cache line that is tracked in the DC's directory. Given an initial condition, coherence transactions can be initiated by the CPU which causes the states in the LLC and DC's directory to diverge briefly till the transaction is completed. Once the transaction is completed, the CPU and FPGA will again converge on the remote state of the cache line. This gives us the following initial conditions:

- State of cache line in CPU's LLC and RS in DC's directory are both I.
- State of cache line in CPU's LLC and RS in DC's directory are both S.
- State of cache line in CPU's LLC and RS in DC's directory are both E/M.

5.3 Maintaining Coherence Invariants

One of the design choices made was to not allow applications on the FPGA to cache FPGA-homed cache lines (design-choice 3.4). The CPU's LLC is the only coherence controller that can issue *coherent caching messages* and coherence controllers on the FPGA interact with the DC through *coherent non-caching messages*. This also means that the CPU is the only writer to copies of cache lines and that state of cache line on the FPGA (HS) is always I. Thus the SWMR invariant is maintained by default.

It is the responsibility of the DC protocol state machine to maintain the *data-value* invariant and provide the most up-to-date copy of an FPGA-homed cache line for caching requests from the CPU. How this can be done by the state space exploration tool is discussed in the following sections.

Takeaway 5.2. *CPU's LLC is the only coherence controller that can write to cached copies of FPGA-homed cache lines. So SWMR invariant is maintained by default and HS of all cache lines are I. This also implies FPGA applications can only perform coherent non-caching operations on FPGA-homed cache lines.*

The steps taken to maintain coherence invariants are as follows. We first identify a set of initial conditions that can occur in the system. Next, for each of these initial conditions, we apply the rules of interaction between the CPU and DC to the DC protocol model and identify *all possible* coherence transactions that can occur based on the initial conditions. This is done in two sub-steps. In the first sub-step, we look at the coherence transactions that can occur if there is no reordering by the interconnect. These coherence transactions form the specfication of the protocol given the initial conditions. In the second sub-step, we identify *all possible* message reorderings that can occur in these coherence transactions by the interconnect. Then we go through each of the coherence transaction (both specification and reordered transactions) to look at the design choices needed to maintain coherence invariants.

By carefully identifying the set of initial conditions and checking each coherence transaction derived from the model for coherence invariants should guarantee that coherence invariants are maintained by the protocol state machine.

5.4 CPU State: I, DC State: I:I

To begin with, we start with a subset of interactions that happen for an FPGA-homed cache line that is Invalid in CPU's cache and is also marked Invalid in the DC's directory (HS:RS is I:I or 1:1). The most up-to-date value for this cache line is in the FPGA memory.

5.4.1 CPU pathways

This case is represented in our model (Figure 4.1) with the CPU being in step 1 and the DC has CPU's step recorded as 1 in its directory. Looking at all possible paths the CPU can take from step 1, the CPU has two options of going up the stairs, from step 1 to step 2, or step 1 to step 3. Interaction rule 5 forbids the CPU from altering its request before receiving a response. Also the CPU cannot come further down the stairs as it is at its lowest step. Thus there can only be three pathways the CPU can take, the CPU can continue remaining in step 1, or it can issue a read-shared (R12) or read-exclusive (R13) and wait for a response from the DC. These pathways are shown in Figure 5.1

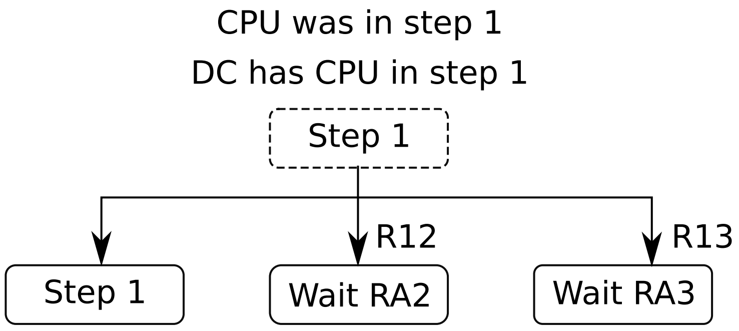


Figure 5.1: *Pathways CPU can take from step 1: The CPU can continue remaining in step 1 or make upgrade requests (R12, R13) and wait for a response from DC.*

When the DC receives the upgrade request, it has to ensure that the coherence invariants are met and respond to the request with the cache line data.

5.4.2 Specification and maintaining coherence invariants

When the cache line is invalid in both home and remote nodes, the most up-to-date copy of the cache line is in the memory of the home node. Thus to maintain *data-value* invariant for CPU's upgrade requests, the DC should read the FPGA memory for the cache line and respond to the CPU. Since the CPU is the only node that can currently read or write to the cache line, SWMR invariant is trivially maintained.

The transactions for both scenarios are shown in Figure 5.2. Upon receiving an upgrade request from the CPU, the DC reads the cache line data from its memory and sends a response to the CPU. The DC also updates the remote state of this cache line in its directory; For read-shared (R12) request, the DC sends an RA2 and updates its remote state to S (1:2), and for the read exclusive (R13) request, the DC sends an RA3 and updates its remote state to E (1:3). The CPU upgrades to the new state upon receiving a response from the DC.

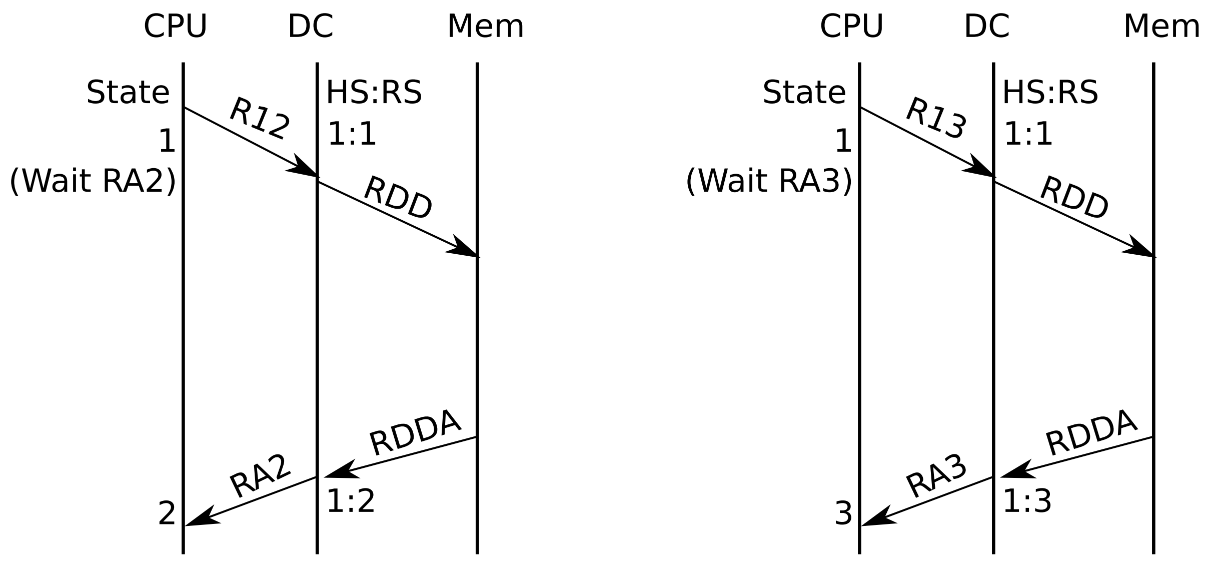


Figure 5.2: *Read-Shared and Read-Exclusive transactions: Since FPGA memory has the most up-to-date value, the DC should read the memory and send its content as a response.*

Note 5.1. *If the HS of a cache line is I, the DC is free to provide exclusive access for a read-shared request from the CPU since the CPU will hold the only copy of the cache line. In this case, the DC can send an RA3 instead of RA2 and this optimization is optional and can differ between platforms. The CPU's native protocol in Enzian allows for this optimization.*

The two transactions in Figure 5.2 can be be represented in terms of the following state equations which forms the specification given the initial conditions (check subsection 4.6.1 for the format of state equations).

$$\begin{aligned} 1:1, \{R12\} &\rightarrow 1:2, (\textit{read memory and Send } RA2) \\ 1:1, \{R13\} &\rightarrow 1:3, (\textit{read memory and Send } RA3) \end{aligned} \tag{5.1}$$

For both equations, the present home and remote states of the the cache line in DC's directory is 1:1. When the DC observes an R12 (or R13), it reads the memory and responds to the CPU with an RA2 (or RA3). The DC finally updates the state of this cache line to from 1:1 to 1:2 (or 1:3) in its directory.

5.4.3 Reordering effects and maintaining coherence invariants

For a cache line, the CPU can issue a single upgrade request before waiting for a response. No other coherence messages can be issued by the CPU for this cache line till the response is received. Since there is only one message in transit, there can be no reordering by the interconnect for these coherence transactions.

5.4.4 Building the state machine

Equations 5.1 can be also be represented in the form of a protocol state machine shown in Table 5.1, where given a present-state and coherence message, the contents of the cell gives the next state and action to be performed by the DC (check subsection 4.7.1 for more details on representing state equations in the form of a a table).

Present HS:RS	R12	R13
1:1	1:2 (read memory and send RA2)	1:3 (read memory and send RA3)

Table 5.1: *DC protocol state machine (aka state table) to handle coherence transactions issued by the CPU for a cache line that is invalid in CPU's LLC.*

Deadlocks in protocol state machine: Given the initial condition the state machine in Table 5.1 only waits for messages that are defined by the specification and there is no dependency between events for handling them. If we assume that the DC protocol model is correct, there are no deadlocks in the protocol state machine. Check subsection 3.4.1 for more information on protocol deadlocks.

Note 5.2. *We assume the model is correct because it works in practice and there is no real way to prove that the model matches the CPU's native protocol specification.*

Performance of protocol state machine: The protocol state machine described above suffers from performance bottleneck due to *latency of the read/write operation.* The state machine requires issuing a request to memory and waiting for a response before it can handle new upgrade requests. This leads to serialization of memory transactions, blocking of coherence transactions and does not allow for multiple outstanding or out-of-order memory transactions. This bottleneck is illustrated by Figure 5.3 where upgrade request on a second cache line is *blocked* till the read operation on the first cache line completes.

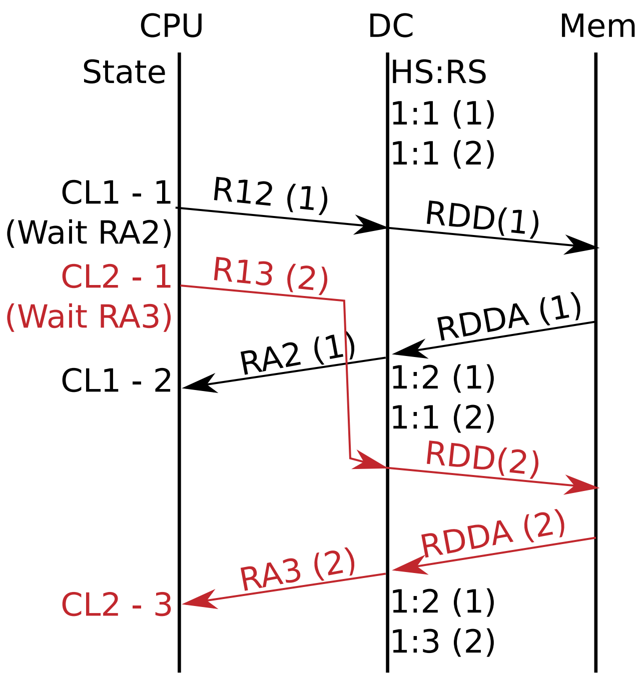


Figure 5.3: *Serializing memory transactions lead to performance bottleneck in the protocol state machine due to blocking of upgrade request for a second cache line by the DC till a read response for the first cache line is received from memory. The cache line corresponding to a coherence event and DC state is shown within parenthesis.*

Since most upgrade requests require reading from the memory, these transactions are in the critical path and can quickly become the performance bottleneck.

The solution to this problem is to have the state machine transition to an *intermediate state* upon issuing the read request and continue handling new transactions. Eventually when the read response arrives, the intermediate state provides context on what is to be done. Thus at the cost of increased state machine size, we can optimize for performance in the read critical path.

This optimization is illustrated by Figure 5.4. When the DC receives a read-shared request for the first cache line, it issues a read request to memory (RDD) and updates its directory with an *intermediate state* (1:2pRA2) for this cache line. The DC is now free to handle the read-exclusive request for the second cache line by issuing an RDD and transitioning to an intermediate state (1:3pRA3). Eventually when read responses from the memory (RDDA) arrives, the intermediate state would dictate the response sent and new states of the cache lines. This also allows the DC to handle transactions that are out of order

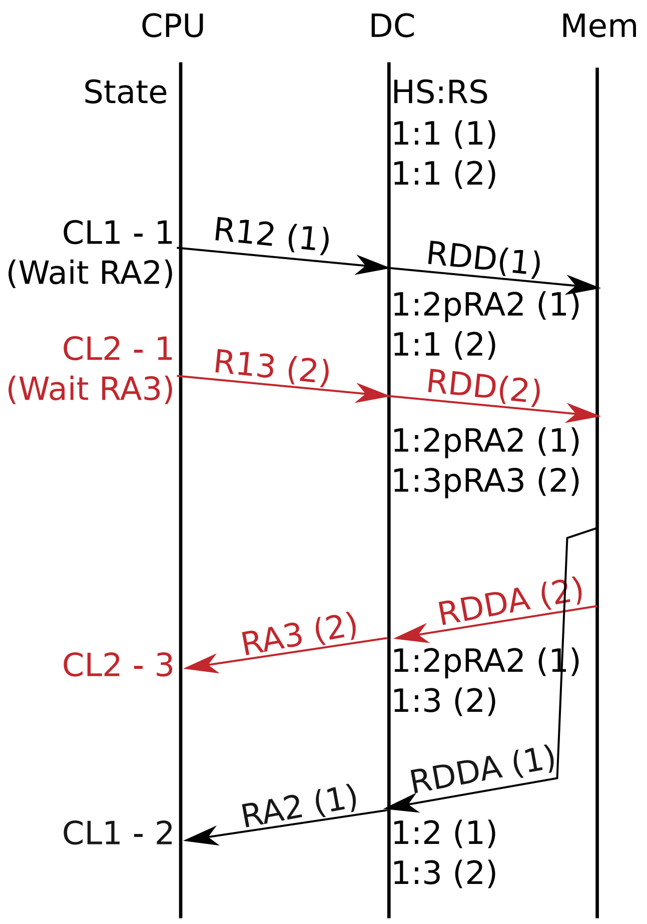


Figure 5.4: *Having the state machine transition to intermediate state prevents blocking and serialization of transactions. It also allows the DC to handle transactions out of order.*

as shown in Figure 5.4 where the read response from the memory for the second cache line is received before the first. first. Note that the names of intermediate remote state 2pRA2 and 3pRA3 are arbitrary with the following convention for naming: 2pRA2 means the final remote state will be 2 (S) pending ('p' in 2pRA2) a response RA2.

Design Choice 5.1. *Serializing read and write transactions can cause performance bottlenecks. Using intermediate states, the memory transactions can be split to handle requests and responses independently and improve performance. The downside of this design choice is that it increases the number of states in the state space.*

Revisiting specification state equations to include performance optimizations: State equations 5.2 show the modifications made to equations 5.1 to allow for this optimization for a single cache line. The event R12 transitions the RS of a cache line from state 1 to intermediate state 2pRA2 and triggers the RDD (memory read request) event. Eventually when the memory response event (RDDA) arrives, the cache line transitions from RS 2pRA2 to state 2. Note that read requests (RDD) are sent to the memory whereas coherence messages (RA2 and RA3) are sent to the CPU.

$$\begin{aligned} 1:1,\{R12\} &\rightarrow 1:2pRA2,(Send\ RDD) \\ 1:2pRA2,\{RDDA\} &\rightarrow 1:2,(Send\ RA2) \\ 1:1,\{R13\} &\rightarrow 1:3pRA3,(Send\ RDD) \\ 1:3pRA3,\{RDDA\} &\rightarrow 1:3,(Send\ RA3) \end{aligned} \tag{5.2}$$

Effect of reordering of coherence events on state equations: All state equations in equation 5.2 have only one coherence event in transit for a cache line and so there are no reordering effects.

Building the state machine: Table Table 5.2 shows the protocol state machine for equations 5.2, where “X” denotes coherence events that are not possible under the constraints of the state equations.

Present HS:RS	R12	R13	RDDA
1:1	1:2pRA2 (Send RDD)	1:3pRA3 (Send RDD)	X
1:2pRA2	X	X	1:2 (Send RA2)
1:3pRA3	X	X	1:3 (Send RA3)

Table 5.2: *DC protocol state machine to handle coherence transactions issued by the CPU for a cache line that is invalid in CPU’s LLC. The state machine is optimized for performance along the read critical path.*

Deadlocks in protocol state machine: By simply observing the state machine in Table 5.2 we can convince ourselves that the state machine only waits for coherence messages that are defined by the specification and that there are no dependencies between events when it comes to handling them. Thus according to subsection 3.4.1 there should not be any protocol deadlocks.

To summarize, we started with an initial condition and used the DC protocol model to identify all pathways the CPU can take given the initial condition. Using these pathways, we identified the coherence transactions (represented in the form of state equations) that form the specification of the protocol given the initial condition.

We then optimized the specification to avoid performance bottlenecks in the critical path. Finally, we looked at all possible re-orderings of the specification coherence transactions (in this case there was only one possible ordering and no conflicts), verified if the coherence invariants are maintained, and built a state machine (or state table). We also visually checked that the state machine adheres to the specification and no protocol deadlocks are possible.

5.5 CPU State: I, DC State: I:S

Now we look at the subset of coherence transactions that can happen when an FPGA-homed cache line is *Shared* in the CPU's LLC which is also reflected in the directory of the DC (HS:RS is I:S or 1:2). To begin with, we have assumed that the DC does not initiate any forward downgrade transactions and this assumption will be removed in the next chapter. Since home state is I and remote state is S the most up-to-date value of the cache line resides in the FPGA memory with a read-only copy in the CPU's LLC.

5.5.1 CPU pathways

The first step is to get the specification of coherence transactions possible under this scenario. For this we need to investigate all possible pathways the CPU can take in the protocol model given the initial conditions.

This case is represented in our model with the CPU being in step 2 and the the DC has CPU's step recorded as 2 in its directory (initial condition). Figure 5.5 shows all possible paths the CPU can take from step 2. Under each pathway is the state equation that the DC would observe given the event-ordering of the CPU is maintained i.e. the specification of the protocol given the initial condition. The pathways are as follows.

① The CPU can continue remaining on step 2 (no coherence message issued to DC). This means the CPU continues to hold a *Shared* copy of the cache line in its LLC.

② The CPU can make a request to go up from step 2 to 3 by issuing upgrade request, R23, and waiting for a response. This corresponds to CPU requesting an upgrade of the *Shared* cache line to *Exclusive*.

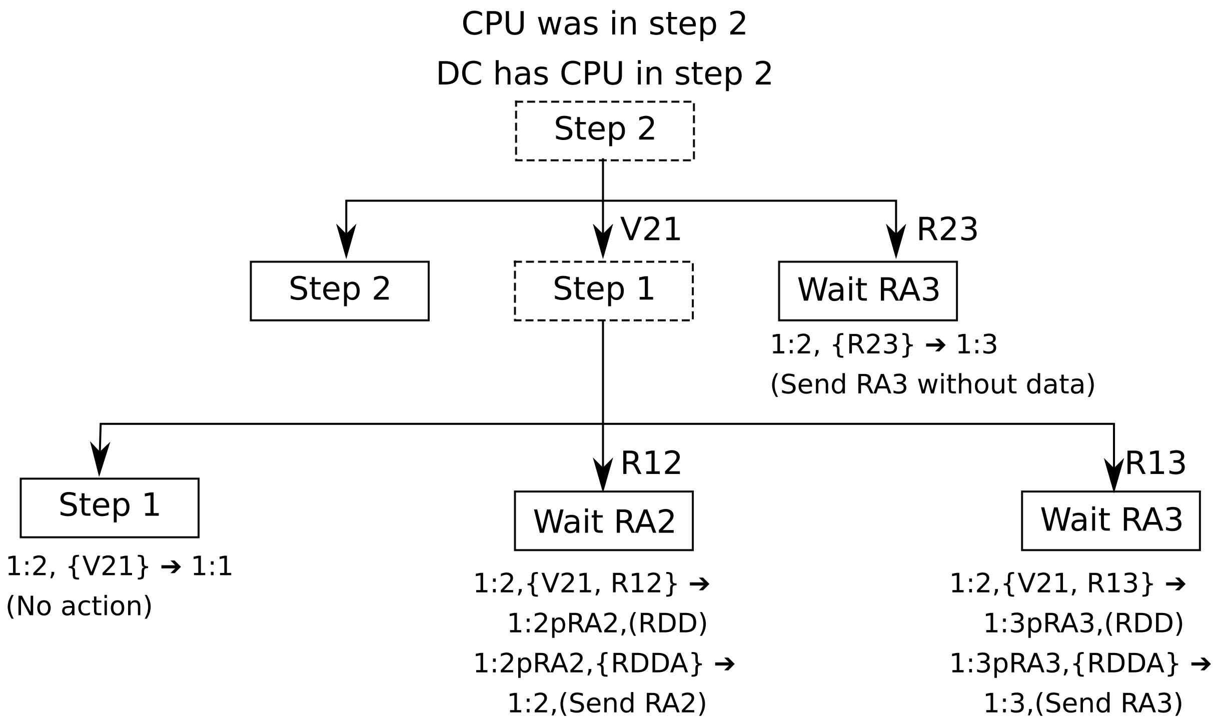


Figure 5.5: *Pathways CPU can take from step 2: remain at step 2 or request to go up from step 2 to 3 or come down to step 1 and then remain at step 1 or request to go up from step 1 to 2 or 3. Coherence message issued by CPU and state equation in each scenario is also shown.*

③ The CPU can voluntarily come down from step 2 to 1 by issuing V21. This corresponds to CPU invalidating its copy of the cache line in its LLC.

④ At step 1, the CPU can choose to remain at step 1 or issue an upgrade request to step 2 (R12) or step 3 (R13) as described in section 5.5.

Given the pathways, lets look at the state equations that describe the pathways. These state equations describe the specification of the protocol given the initial conditions.

5.5.2 Specification and maintaining coherence invariants

The current design choices do not allow the FPGA to make caching requests on a cache line. This means the home state of the cache line is always I. Thus to maintain SWMR invariant for pathway ② the CPU requests an upgrade from shared to exclusive (R23), the DC does not have to perform any invalidations on the home node before issuing the upgrade response. Furthermore, the DC does not have to respond with the cache line data since the CPU already has the most up-to-date copy of the cache line in its LLC. This guarantees *data-value* invariant and eliminates the need for a DC initiated memory-read transaction. Thus the DC state equation for this pathway is

$$1:2, \{R23\} \rightarrow 1:3, (Send\ RA3\ without\ data) \tag{5.3}$$

Deviating a bit from writing the specification, it should be noted that specification is written by us and can be altered according to our needs. For example, read requests from the DC can be used as a notification signal by applications on the FPGA. A common concern is that the upgrade request from S to E (shown in equation 5.3) does not generate a read request and thus does not signal the application at all. It is at this point the reader should not despair and remember that the state equations are written by us and can be altered to generate the read request signal. For example equation 5.4 shows how equation 5.3 can be changed to generate a read request.

$$\begin{gathered} 1:2, \{R23\} \rightarrow 1:2pRA3nod, (Send\ RDD) \\ 1:2pRA3nod, \{RDDA\} \rightarrow 1:3, (Send\ RA3\ without\ data) \end{gathered} \tag{5.4}$$

This comes with the responsibility that the new state equation should guarantee that the coherence invariants are met, don't leads to deadlocks and is performant. For now we continue to have equation 5.3 in our specification instead of equation 5.4 and continue with the next pathway.

Takeaway 5.3. *The DC specification state equations are written by us. As a result, we have control over what each coherence transaction should do. We can break up coherence transactions to make them intercept-able at the application level or add new signaling interfaces with different properties. The only responsibility being that all guarantees required of coherence transactions at the DC protocol must be provided by our state equation as well. This can be very useful to reduce the burden of applications that interact with the DC.*

For pathway ③, when the CPU invalidates its *Shared* copy of a cache line (by issuing V21), the FPGA memory continues to have the most up-to-date version of the cache line. Thus the DC does not have to perform any action to maintain coherence invariants other than accounting for the voluntary downgrade coherence message. This gives the DC state equation

$$1 : 2, \{V21\} \to 1 : 1, (No\ action) \tag{5.5}$$

For pathway $\textcircled{4}$, the CPU issues a voluntary downgrade (V21) followed by an upgrade request (R12 or R13). In order to get the specification state equation, we consider the coherence transaction in which the DC receives these coherence events in the same order (see linearizablity for specification in subsection 4.6.2). Once the voluntary downgrade response is received by the DC, the remote state of the cache line in its directory transitions from S to I. So when the upgrade request is received, the DC behaves exactly as it did in section 5.4 for which we have the specification in equation 5.2. Thus the DC state equations for this pathway are as follows.

$$\begin{aligned}
1 : 2, \{V21, R12\} &\to 1 : 2pRA2, (RDD)\\
1 : 2pRA2, \{RDDA\} &\to 1 : 2(SendRA2)\\
1 : 2, \{V21, R13\} &\to 1 : 3pRA3, (RDD)\\
1 : 3pRA3, \{RDDA\} &\to 1 : 3(SendRA3)
\end{aligned} \tag{5.6}$$

These state equations combined with the specification from section 5.4 form the specification of the protocol given the initial conditions. We have also seen how coherence invariant is maintained for all state equations in the specification. Next lets look at the state equations that arise due to reordering of coherence events in the specification.

5.5.3 Reordering effects and maintaining coherence invariants

Conflicts due to message reordering: For pathways $\textcircled{1}$ to $\textcircled{3}$ there is only one message involved in the coherence transaction so there are no conflicts due to message reordering by the interconnect. For pathway $\textcircled{4}$ when there are two messages in transit (voluntary downgrade response and upgrade request), the messages may be reordered by the interconnect. The *scrambler* (subsection 4.3.1) in the directory protocol model predicts that these two messages can be reordered in 2! ways. Thus the DC can receive either the voluntary downgrade first followed by the upgrade request or the upgrade request first followed by the voluntary downgrade.

In the latter scenario, conflicts arise when the DC observes an upgrade request from I to S or E (R12 or R13) when the remote state of this cache line is still S in its directory. Under such conditions, it is the responsibility of the DC to infer that the state of the cache line is currently I in the CPU's LLC, a voluntary downgrade message from S to I is in transit and to maintain coherence invariants.

Takeaway 5.4. *Previously in takeaway 4.3 we have seen conflict scenarios can arise due to latency of the interconnect which is handled by conflict responses. The second source of conflict scenarios is reordering of coherence messages by interconnect as described here. In this case, based on the coherence event received, the DC should infer the actual state of the cache line in the CPU's LLC and assume some other coherence events are in transit. Interconnect latency and message reordering are the only sources of conflicts in our system.*

For pathway ④ *with* reordering conflicts where the DC observes an upgrade request before the downgrade response, based on the state of the cache line in its directory, the DC can infer that a voluntary downgrade message from S to I is in transit. Since the CPU invalidated a read-only copy of the cache line, the downgrade message does not have any dirty data and FPGA memory has the only copy of the cache line. Thus to maintain *data-value* invariant and service upgrade requests, the DC is free to read the FPGA memory and serve the request as long as it can account for the voluntary downgrades that are in transit. Thus the state equations are:

$$\begin{aligned}1:2,\{R12,V21\} &\rightarrow 1:2pRA2,(RDD)\\ 1:2pRA2,\{RDDA\} &\rightarrow 1:2(SendRA2)\\ 1:2,\{R13,V21\} &\rightarrow 1:3pRA3,(RDD)\\ 1:3pRA3,\{RDDA\} &\rightarrow 1:3(SendRA3)\end{aligned} \tag{5.7}$$

Given the two-message DC state equations without any reordering in equation 5.6, the reader can convince themselves that state equations in equations 5.6 and 5.7 cover all 2! possible reordering scenarios and their corresponding state equations.

Accounting for messages in transit under conflict scenarios: In order to keep track of the aforementioned voluntary downgrades in transit, intermediate states can be used as shown in Figure 5.6 and described below.

The CPU invalidates a shared cache line by issuing V21 and follows it up with an upgrade request from I to S state (R12). For some reason this V21 is delayed by the interconnect.

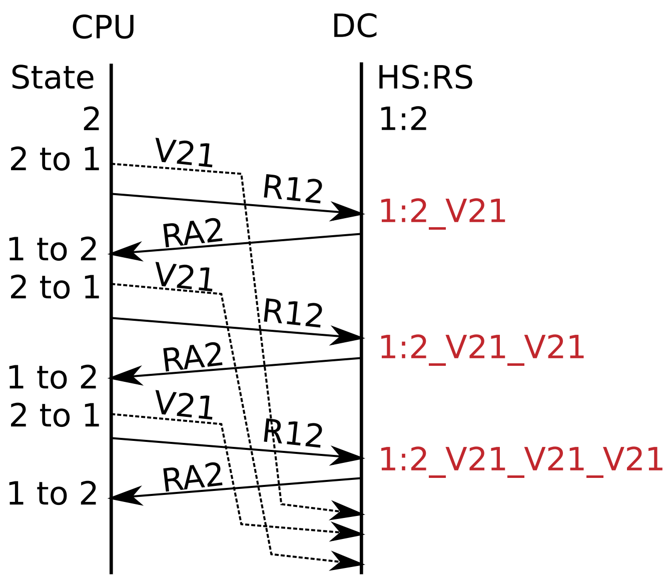


Figure 5.6: *The voluntary downgrade (V21) sent before the upgrade request (R12) by the CPU is delayed by the interconnect causing a conflict. This conflict is resolved by creating an intermediate RS (2_V21) indicating that a voluntary downgrade is in transit. Since there is no upper bound on the number of times this conflict can occur, adding a new state every time will lead to state space explosion.*

The DC has the cache line in state 1:2 when it receives the first R12 upgrade request. The DC can infer that a voluntary downgrade is in transit and transition to intermediate state $1:2\_V21$ (meaning: state of cache line in CPU will be 2 but a V21 is in transit) and respond to the upgrade request with data from the FPGA memory. If this cycle is repeated multiple times with the downgrade message delayed every time, the number of states that are required to keep track of voluntary downgrades also increases which can lead to explosion on the number of intermediate states as shown in Figure 5.6. One way to avoid this is limit the number of times the cycle is allowed to happen by *stalling* any upgrades and waiting for the downgrade messages. In order to simplify the state machine (and in the future, to maintain coherence invariants when dirty data is involved) we make the choice of stalling requests at the first sign of conflict in the state space exploration tool. Although this affects performance, these conflict scenarios are not expected to happen often and thus are not in the critical path.

Design Choice 5.2. *When faced with a conflict due to the interconnect reordering coherence messages, stall requests till all inferred responses in transit are received. Another options is to consume the request into an intermediate state instead of stalling it and de-*

lay responding to the request until all in-flight responses have arrived. The first option was chosen but in hindsight, the second option could have simplified DC implementation. Stalling an event in its VC causes head-of-line blocking which can be avoided with the second approach but this statement needs to be explored and verified.

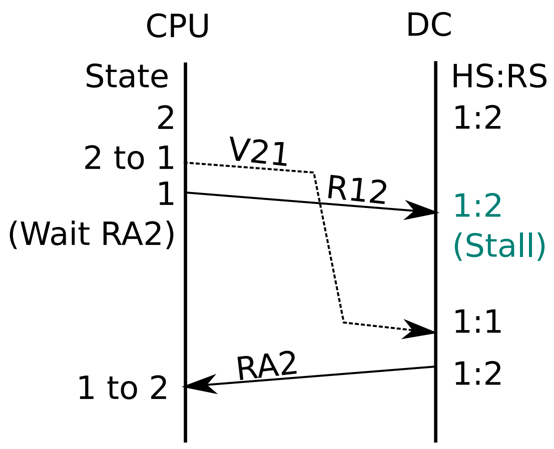


Figure 5.7: *Avoiding state explosion by stalling requests (R12) till all responses in transit (V21)are received. The request is not consumed into an internal state but rather continues to sit at the head of its VC.*

Figure 5.7 illustrates this design choice. The DC has the cache line in state 1 : 2 when it receives R12. Since this is the upgrade request from step 1 to 2, the DC can detect a conflict, infer that the CPU is currently in step 1, and that a V21 downgrade message is in transit. The DC then stalls the upgrade request till all responses in transit are received and then responds to the upgrade request. This gives us the following state equations.

$$\begin{aligned} 1:2, \{R12\} &\rightarrow 1:2, (Stall) \\ 1:2, \{R13\} &\rightarrow 1:3, (Stall) \end{aligned} \tag{5.8}$$

Takeaway 5.5. *Conflicts due to reordering of coherence events is handled by the DC through a combination of stall operations and having intermediate states.*

Deadlocks due to stalling: Stalling coherence events across message classes can lead to deadlocks. For example, consider a scenario (shown in Figure 5.8) where both a request $Req1$ in request channel and a response $Rsp1$ in response channel need response $Rsp2$ to be handled. If both request and response channels are stalled, $Rsp2$ never reaches the state machine due to head-of-line blocking and causes a deadlock.

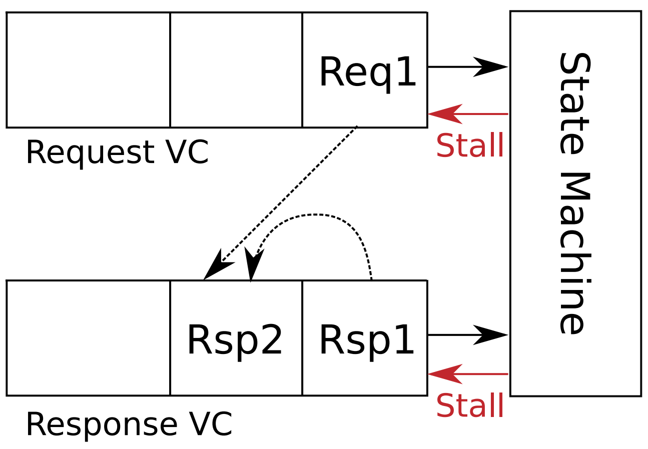


Figure 5.8: *Stalling both requests and responses can cause deadlocks: Req1 and Rsp1 are both stalled waiting for Rsp2 to be handled but Rsp2 is stuck between Rsp1 in the response VC.*

In order to avoid this we allow the state space exploration tool to stall only requests and never responses. Responses will always be consumed into an internal state and the action corresponding to the event will be performed immediately. Applying this rule to previous example, $Req1$ can be stalled but $Rsp1$ should be popped off the channel and consumed into a state in the state machine. This brings $Rsp2$ to the top of the channel allowing the state machine to eventually handle $Req1$.

Design Choice 5.3. *The DC protocol state machine can only stall messages of request message-classes. Messages in the response message-classes can never be stalled and must be consumed into an internal state to avoid deadlocks. Any action associated with the response event will also be performed immediately. This is more of a design necessity than choice.*

5.5.4 Building the state machine

Earlier (design-choice 4.2) we made the decision that the state machine handles only one coherence event at a time. Building the state machine for state equations with a single coherence event (equations 5.3 and 5.5) is straight forward as shown in subsection 4.7.1. Now let us see how to build the state machine for state equations with more than one coherence event, specifically equations in 5.6 and 5.7

Let us consider the following state equation from 5.6 (again repeated in equation 5.9) with two messages where the present state of a cache line is 1:2 in the DC's directory. Here the voluntary downgrade V21 arrives first and needs to be handled by the state machine. The question is what is next state and action to be performed by the state machine in this scenario.

$$\begin{aligned} 1:2, \{\mathbf{V21}, R12\} &\rightarrow 1:2pRA2, (RDD) \qquad \text{spec 5.6} \\ 1:2, \{\mathbf{V21}\} &\rightarrow \textit{next state}?, (\textit{action}?) \end{aligned} \tag{5.9}$$

State equation 5.5 already gives us an answer to this question.

$$1:2, \{V21\} \rightarrow 1:1, (\textit{No action}) \tag{5.10}$$

Substituting (see substitution operator in subsubsection 4.7.1.1) equations 5.10 into 5.9 allows us to infer new state equations with fewer coherence events.

$$\begin{aligned} 1:2, \{V21, R12\} &\rightarrow 1:2pRA2, (RDD) && \text{spec 5.6} \\ 1:2, \{V21\} &\rightarrow 1:1, (\textit{No action}) && \text{known from 5.5} \\ 1:1, \{R12\} &\rightarrow 1:2pRA2, (RDD) && \text{inferred by substitution} \\ & && \text{consistent with 5.2} \end{aligned} \tag{5.11}$$

The inferred state equation can either give us a brand new state equation or it should be consistent with a previously known state equation. In this case the inferred state equation is consistent with the previously defined equation 5.2. Thus through substitution, we have broken down a 2-message state equation into two 1-message state equations.

Applying the same reasoning for the next 2-message state equation in 5.6, we get:

$$\begin{aligned} 1:2, \{V21, R13\} &\rightarrow 1:3pRA3, (RDD) && \text{spec 5.6} \\ 1:2, \{V21\} &\rightarrow 1:1, (\textit{No action}) && \text{known from 5.5} \\ 1:1, \{R13\} &\rightarrow 1:3pRA2, (RDD) && \text{inferred by substitution} \\ & && \text{consistent with 5.2} \end{aligned} \tag{5.12}$$

Now lets look at the 2-message state equations where an upgrade request is received before the voluntary downgrade response as seen in equation 5.7. In these scenarios the CPU issues a voluntary downgrade followed by an upgrade request which is then reordered by the interconnect before being observed by the DC. The state machine in this case stalls (for stall operator see subsubsection 4.7.1.2) the upgrade request till the voluntary downgrade response is received (design-choice 5.2). Thus we have the following equations:

$$\begin{aligned}
1:2,\{R12,V21\} &\rightarrow 1:2pRA2,(RDD) && \text{spec 5.7}\\
1:2,\{R12\} &\rightarrow 1:2,(\mathit{Stall}) && \text{known from 5.8}\\
1:2,\{V21,R12\} &\rightarrow 1:2pRA2,(RDD) && \text{order after stalling}\\
& && \text{consistent with 5.6}\\
1:2,\{R13,V21\} &\rightarrow 1:3pRA3,(RDD) && \\
1:2,\{R13\} &\rightarrow 1:3,(\mathit{Stall}) && \text{known from 5.8}\\
1:2,\{V21,R13\} &\rightarrow 1:3pRA3,(RDD) && \text{order after stalling}\\
& && \text{consistent with 5.6}
\end{aligned} \tag{5.13}$$

It can also be seen that stalling a coherence event by the state machine is akin to reordering the state equation to get the intended order in which the events were issued by the CPU and that no new state equations are created.

Now that we have solved all the multiple message state equations we have the following single message state equations shown in equation 5.14. This can be used to build the state machine shown in Table 5.3.

$$\begin{aligned}
1:1,\{R12\} &\rightarrow 1:2pRA2,(RDD) && \text{from 5.2}\\
1:2pRA2,\{RDDA\} &\rightarrow 1:2,(\mathit{Send\ RA}2) && \text{from 5.2}\\
1:1,\{R13\} &\rightarrow 1:3pRA3,(\mathit{Send\ RDD}) && \text{from 5.2}\\
1:3pRA3,\{RDDA\} &\rightarrow 1:3,(\mathit{Send\ RA}3) && \text{from 5.2}\\
1:2,\{R23\} &\rightarrow 1:3,(\mathit{Send\ RA}3\ \mathit{no\ data}) && \text{from 5.3}\\
1:2,\{V21\} &\rightarrow 1:1,(\mathit{No\ action}) && \text{from 5.5}\\
1:2,\{R12\} &\rightarrow 1:2,(\mathit{Stall}) && \text{from 5.8}\\
1:2,\{R13\} &\rightarrow 1:3,(\mathit{Stall}) && \text{from 5.8}
\end{aligned} \tag{5.14}$$

Deadlocks in protocol state machine: Given the initial condition the state machine in Table 5.3 only waits for messages that are defined by the specification. Furthermore the dependency between requests and response messages will not cause a deadlock because these messages arrive in different VCs and response messages are always *sunk* (consumed into an internal state) by the state machine. This guarantees that the state machine is deadlock free.

Present HS:RS	R12	R13	R23	V21	RDDA
1:1	1:2pRA2 (Send RDD)	1:3pRA3 (Send RDD)	X	X	X
1:2pRA2	X	X	X	X	1:2 (Send RA2)
1:3pRA3	X	X	X	X	1:3 (Send RA3)
1:2	1:2 (Stall)	1:2 (Stall)	1:3 (Send RA3 no data)	1:1 (No action)	X

Table 5.3: *DC protocol state machine (aka state table) to handle coherence transactions issued by the CPU for a cache line that is shared or invalid in CPU's LLC.*

Performance of protocol state machine: The read critical path has already been optimized in section 5.4 which gets carried over to this section.

5.6 CPU state: E, DC State: I:E

Now we look at the subset of coherence transactions that can happen when an FPGA-homed cache line is *Exclusive* in the CPU's LLC which is also reflected in the directory of the DC (HS:RS is I:E or 1:3). As mentioned before, we assume no forward-downgrade transactions are initiated by the DC. Since home state is I, remote state is E, and upgrade from E to M in CPU's LLC is silent, it is safe to assume that the most up-to-date value of the cache line resides in the CPU's LLC and the copy of the cache line in the FPGA memory is stale.

First we investigate all pathways the CPU can take under the protocol model, given these initial conditions. These pathways would be used to identify the state equations that form the specification of the protocol.

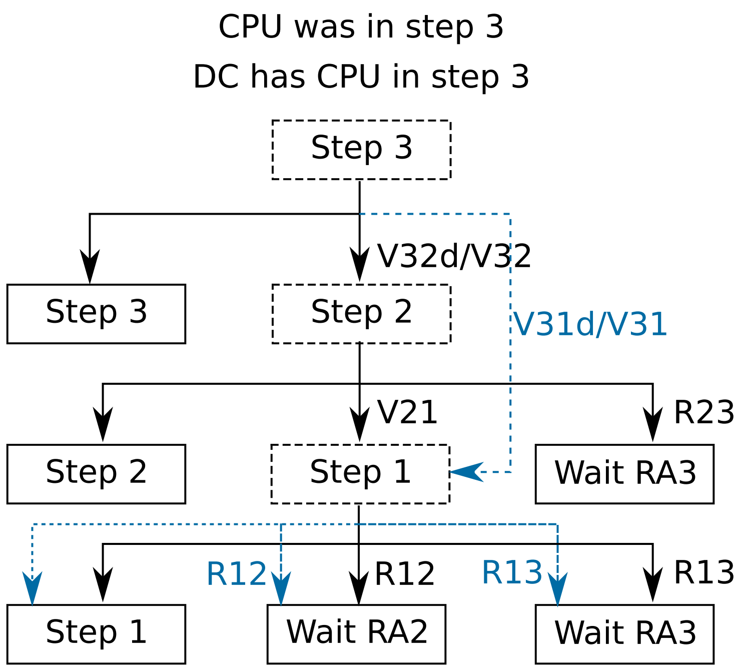


Figure 5.9: *Pathways CPU can take from step 3: remain at step 3, or come down to step 2, or come down to step 1 (shown in a different color). At step 2: remain at step 2 or request to go up from step 2 to 3 or come down to step 1. At step 1: remain at step 1 or request to go up from step 1 to 2 or 3. Coherence message issued by CPU and state equation in each scenario is also shown.*

5.6.1 CPU pathways

Figure 5.9 shows all pathways the CPU can take at step 3 in our protocol model. It is to be noted that step 3 is the highest step in the model and the CPU cannot go further up the stairs. The remaining pathways are described below:

1. The CPU can continue remaining in step 3 and no coherence message is issued to the DC. This means the CPU continues to hold an exclusive/modified copy of the cache line in its LLC.

2. The CPU can voluntarily come down from step 3 to step 2 and it does so by issuing a V32 if data in CPU's LLC is clean, or V32d if the data is modified in the CPU's LLC. This corresponds to CPU cleaning its copy of the cache line and continuing to hold a read-only copy of the cache line by transitioning from *exclusive* to *shared*.

③ The CPU alternatively can voluntarily come down from step 3 to step 1 and it does so by issuing a V31 (if data is clean) or V31d (if data is dirty). This corresponds to the CPU clean-invalidating its copy of the cache line by transitioning from *exclusive* to *invalid*, cleaning any dirty data in the process. The CPU does not hold any copy of the cache line after issuing the downgrade message.

④ Once the CPU is at step 2 or step 1, the pathways are exactly the same as the ones discussed in section 5.5 and section 5.4.

Next we identify the specification state equations that are introduced by this initial condition, which will also be added to the growing specification list.

5.6.2 Specification and maintaining coherence invariants

For pathway ②, the DC has the home state of the cache line as I and remote state as E (initial state is 1:3) when it receives the voluntary downgrade message (V32 or V32d). If the voluntary downgrade message has dirty data, it has to be written back to the FPGA memory before transitioning the remote state from E to S. Since write operation is a high latency operation on a critical path, having an additional state to transition to after issuing a write request (WDD) will avoid serialization and allow for multiple outstanding write transactions, thereby improving performance. This gives us the following state equations, V32d is the coherence event that generates a write request (WDD) which is implicit in the state equation, the write response (WDDA) arrives eventually.

$$\begin{aligned} 1:3, \{V32\} &\rightarrow 1:2, (\textit{No action}) \\ 1:3, \{V32d, WDDA\} &\rightarrow 1:2, (\textit{No action}) \end{aligned} \tag{5.15}$$

It is to be noted that the voluntary downgrade response with dirty data is of the response message class and would have to be handled immediately (without stalling design-choice 5.3). This means there is an implicit write request issued by the DC to the memory, as soon as the downgrade response is handled and the state machine would have to transition to an intermediate state. This implicit write action (WDD) is not shown in the state equation and is initiated as soon as the coherence event is received, but the response from memory is part of the state equation. Since in the specification we treat memory operation on a cache line as atomic, the memory response immediately follows the coherence event that generates the memory request.

Following a downgrade from step 3 to 2, the DC might receive an upgrade request from step 2 to 3. Since the CPU already has the most up-to-date shared-copy of the cache line and is the only controller that can have the cache line in exclusive, it is sufficient if the DC responds to the upgrade request without data after accounting for the voluntary downgrade response to maintain *data-value* invariant. This is shown in equation 5.16 shown below.

$$\begin{gathered}1:3,\{V32,R23\}\rightarrow 1:3,(Send\ RA3\ no\ data)\\ 1:3,\{V32d,WDDA,R23\}\rightarrow 1:3,(send\ RA3\ no\ data)\end{gathered}\tag{5.16}$$

Alternatively, following the downgrade message from step 3 to 2, the DC can receive another voluntary downgrade from step 2 to 1. The DC issues a write request (WDD) as soon as the voluntary downgrade V32d is received.

$$\begin{gathered}1:3,\{V32,V21\}\rightarrow 1:1,(No\ action)\\ 1:3,\{V32d,WDDA,V21\}\rightarrow 1:1,(No\ action)\end{gathered}\tag{5.17}$$

As previously discussed, the DC can receive upgrade requests from step 1 to 2 or step 1 to 3 once voluntary downgrades from step 3 to 2 and step 2 to 1 are received. Since the CPU does not have a copy of the cache line at this point, any dirty data that was previously sent by the CPU must be written to the memory (for consistency) before reading it back and sending a response. Writing dirty data into the memory ensures that the FPGA memory has the most up-to-date value, which can be read back and issued as a response to the upgrade request. This ensures the upgrade requests get the most up-to-date copy of the cache line from the FPGA memory thereby holding *data-value* invariant.

$$\begin{gathered}1:3,\{V32,V21,R12\}\rightarrow 1:2pRA2,(RDD)\\ 1:3,\{V32d,WDDA,V21,R12\}\rightarrow 1:2pRA2,(RDD)\\ 1:2pRA2,\{RDDA\}\rightarrow 1:2(SendRA2)\end{gathered}\tag{5.18}$$

$$\begin{gathered}1:3,\{V32,V21,R13\}\rightarrow 1:3pRA3,(RDD)\\ 1:3,\{V32d,WDDA,V21,R13\}\rightarrow 1:3pRA3,(RDD)\\ 1:3pRA3,\{RDDA\}\rightarrow 1:3(SendRA3)\end{gathered}\tag{5.19}$$

For pathway $\textcircled{3}$, the DC receives a voluntary downgrade from step 3 to 1 with or without dirty data (V31/V31d). The dirty data in this case has to be written back to memory.

$$\begin{aligned} 1:3, \{V31\} &\rightarrow 1:1, (No\ action) \\ 1:3, \{V31d, WDDA\} &\rightarrow 1:1, (No\ action) \end{aligned} \tag{5.20}$$

Following the voluntary downgrade from step 3 to 1, the DC can again receive an upgrade from step 1 to 2 or step 1 to 3. The DC has to ensure that any dirty data is written back before reading the memory to service these upgrade requests to maintain *data-value* invariant.

$$\begin{aligned} 1:3, \{V31, R12\} &\rightarrow 1:2pRA2, (RDD) \\ 1:3, \{V31d, WDDA, R12\} &\rightarrow 1:2pRA2, (RDD) \\ 1:2pRA2, \{RDDA\} &\rightarrow 1:2(SendRA2) \end{aligned} \tag{5.21}$$

$$\begin{aligned} 1:3, \{V31, R13\} &\rightarrow 1:3pRA3, (RDD) \\ 1:3, \{V31d, WDDA, R13\} &\rightarrow 1:3pRA3, (RDD) \\ 1:3pRA3, \{RDDA\} &\rightarrow 1:3(SendRA3) \end{aligned} \tag{5.22}$$

The state equations above specify what coherence messages the DC can enc outer given the state of the cache line is 1 : 3. Now let us look at all possible re-orderings of these state equations by the interconnect.

5.6.3 Reordering effects and maintaining coherence invariants

We begin by looking at equation 5.15. The first state equation has only one coherence event and so there are no re-orderings possible for this state equation. The second state equation has two coherence events but the the memory write response event (WDDA) will always arrive after the memory write request issued as a consequence of the voluntary downgrade with data. Thus there can be no reordering of the second state equation as well.

Takeaway 5.6. *Memory write requests are always initiated by a voluntary downgrade with dirty data, memory write responses cannot arrive before the coherence event that generates the memory requests.*

Next we look at equation 5.16. The first state equation has two coherence messages which can be reordered in 2! ways: The voluntary downgrade arrives before the upgrade request or the upgrade request arrives before the voluntary downgrade. In the second state equation, the write response (WDDA) will always follow the voluntary downgrade with data (V32d) that initiates the write request. Thus this equation can only be reordered on 2! ways. The reordered state equations are shown below.

$$\begin{aligned} 1:3, \{R23, V32\} &\rightarrow 1:3, (Send\ RA3\ no\ data) \\ 1:3, \{R23, V32d, WDDA\} &\rightarrow 1:3, (send\ RA3\ no\ data) \end{aligned} \tag{5.23}$$

When the upgrade request arrives before the voluntary downgrade from exclusive, there can be dirty data in transit making the contents of the FPGA memory stale. As such, serving the upgrade request immediately upon arrival by reading the FPGA memory would violate the *data-value* invariant. Here design choice (design-choince 5.2) made earlier to stall requests till all inferred responses in transit are received helps with maintaining the *data-value* invariant. Thus we have

$$1:3, \{R23\} \rightarrow 1:3, (Stall) \tag{5.24}$$

In equation 5.17 again there are two coherence events in transit which can be reordered in 2! ways, the voluntary downgrade from step 3 to 2 arrives before voluntary downgrade from step 2 to 1, or voluntary downgrade from step 2 to 1 arrives before voluntary downgrade from step 3 to 2. Coherence invariants are maintained irrespective of the order in which the response messages are handled. The reordered equations are

$$\begin{aligned} 1:3, \{V21, V32\} &\rightarrow 1:1, (No\ action) \\ 1:3, \{V21, V32d, WDDA\} &\rightarrow 1:1, (No\ action) \end{aligned} \tag{5.25}$$

Equation 5.18 has two state equations with three coherence messages in transit which can be reordered in 3! (6) ways. As described before, upgrade requests are stalled till all voluntary downgrade messages in transit are handled to maintain coherence invariants.

The reordered state equations are

$$\begin{aligned}
1:3,\{V32,R12,V21\} &\rightarrow 1:2pRA2,(RDD)\\
1:3,\{V21,V32,R12\} &\rightarrow 1:2pRA2,(RDD)\\
1:3,\{V21,R12,V32\} &\rightarrow 1:2pRA2,(RDD)\\
1:3,\{R12,V32,V21\} &\rightarrow 1:2pRA2,(RDD)\\
1:3,\{R12,V21,V32\} &\rightarrow 1:2pRA2,(RDD)\\
1:3,\{V32d,WDDA,R12,V21\} &\rightarrow 1:2pRA2,(RDD)\\
1:3,\{V21,V32d,WDDA,R12\} &\rightarrow 1:2pRA2,(RDD)\\
1:3,\{V21,R12,V32d,WDDA\} &\rightarrow 1:2pRA2,(RDD)\\
1:3,\{R12,V32d,WDDA,V21\} &\rightarrow 1:2pRA2,(RDD)\\
1:3,\{R12,V21,V32d,WDDA\} &\rightarrow 1:2pRA2,(RDD)
\end{aligned} \tag{5.26}$$

Similarly equation 5.19 also has two state equations with three coherence messages that can be reordered in the following 3! ways. Note that stalling upgrade requests till all voluntary downgrades are handled is enough to maintain *data-value* invariant.

$$\begin{aligned}
1:3,\{V32,R13,V21\} &\rightarrow 1:3pRA3,(RDD)\\
1:3,\{V21,V32,R13\} &\rightarrow 1:3pRA3,(RDD)\\
1:3,\{V21,R13,V32\} &\rightarrow 1:3pRA3,(RDD)\\
1:3,\{R13,V32,V21\} &\rightarrow 1:3pRA3,(RDD)\\
1:3,\{R13,V21,V32\} &\rightarrow 1:3pRA3,(RDD)\\
1:3,\{V32d,WDDA,R13,V21\} &\rightarrow 1:3pRA3,(RDD)\\
1:3,\{V21,V32d,WDDA,R13\} &\rightarrow 1:3pRA3,(RDD)\\
1:3,\{V21,R13,V32d,WDDA\} &\rightarrow 1:3pRA3,(RDD)\\
1:3,\{R13,V32d,WDDA,V21\} &\rightarrow 1:3pRA3,(RDD)\\
1:3,\{R13,V21,V32d,WDDA\} &\rightarrow 1:3pRA3,(RDD)
\end{aligned} \tag{5.27}$$

State equations in equation 5.20 has only one coherence message and so no re-orderings are possible. State equations in equation 5.21 and 5.22 have two coherence messages which can be reordered in 2! ways, the following equations show the reordered state equations. Again, stalling requests till in-transit response messages are received is sufficient to maintain *data-value* invariant.

$$\begin{aligned} 1:3, \{R12, V31\} &\rightarrow 1:2pRA2, (RDD) \\ 1:3, \{R12, V31d, WDDA\} &\rightarrow 1:2pRA2, (RDD) \\ 1:2pRA2, \{RDDA\} &\rightarrow 1:2(SendRA2) \end{aligned} \tag{5.28}$$

$$\begin{aligned} 1:3, \{R13, V31\} &\rightarrow 1:3pRA3, (RDD) \\ 1:3, \{R13, V31d\} &\rightarrow 1:3pRA3, (RDD) \\ 1:3pRA3, \{RDDA\} &\rightarrow 1:3(SendRA3) \end{aligned} \tag{5.29}$$

From all the reordered state equations, it can be seen that maintaining *data-value* invariant requires that upgrade requests be stalled till all downgrade responses in transit are handled. This ensures that any dirty data in transit is written back to the memory before an upgrade request is handled.

$$\begin{aligned} 1:3, \{R12\} &\rightarrow 1:3, (Stall) \\ 1:3, \{R13\} &\rightarrow 1:3, (Stall) \end{aligned} \tag{5.30}$$

Takeaway 5.7. *The design choice 5.2 to stall requests till all in-flight responses are received helps maintain* data-value *invariant by ensuring any dirty data in transit gets written back to memory before the upgrade requests are handled. This ensures that when the memory is read for the upgrade request, the most up-to-date copy is in the memory and not in transit.*

Now we have the specification state equations and all possible re-orderings of these equations that can be observed by the DC. The next step is to build the state machine.

5.6.4 Building the state machine

In this section, we will try to develop an algorithm that will solve the newly identified state equations. From equation 5.14 we have state equations from the CPU being in step 2 and step 1. We will consider them known state equations as they would have to hold even when the CPU steps down from step 3.

We start with the two single-event state equations from the current set (equations 5.15 and 5.20) and since they do not have any re-orderings they can be directly added to the known set of state equations. Thus the following equations will be added to the known set of state equations.

$$\begin{aligned} &1:3,\{V32\} \rightarrow 1:2,(No\ action) \quad \text{add to known} \\ &1:3,\{V31\} \rightarrow 1:1,(No\ action) \quad \text{add to known} \end{aligned} \tag{5.31}$$

Algorithm 4 gives the pseudo-code of adding a single-event state equation to the known set of state equations. The algorithm checks if the newly added state equation is *consistent* with the prescriptions of previously known state equations to ensure that there are no problems with the specification (check 4.7.1.1 for more details on consistency of state equations). When the state equation is consistent, it gets added to the known set of state equations.

Algorithm 4 Adding a new state equation with cardinality of 1 to a set of known state equations.

```
    Input $Eqn : S1, \{M1\} \rightarrow S2, (Action1),$ known
    Output known($known\ set\ of\ state\ equations$)
1: procedure FILL_STATE_1($known, S1, M1, S2, Action1$)
2:     res ← $check\_se\_consistency$(known, S1, M1, S2, Action1)      ▷ Algorithm 2
3:     if res is ‘‘inconsistent’’ then
4:         exit                                                  ▷ Problems in specification
5:     known[(S1,M1)]]["next_state"] ← S2
6:     known[(S1,M1)]]["action"] ← Action1
7:     return(known)
```

Next we look at two-event state equations, we iterate through all two-event state equations in our specification list, identify all possible re-orderings and apply the state equation reduction operators to split the multiple-event state equation to single-event state equations.

The first two-event state equation and its re-orderings are

$$\begin{aligned} &1:3,\{V32d, WDDA\} \rightarrow 1:2,(No\ action) \quad \text{specification} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{No reorderings} \end{aligned} \tag{5.32}$$

We have seen that it is advantageous to split latency sensitive write operation by transitioning to an intermediate state. This is done since write optimization is in the critical path and we do not want to serialize on write transactions.

Receiving a V32d requires a write action to be performed but the state to transition to after performing the write action is not present in the known set of transitions. Thus we create an intermediate state and infer a new single-event state equation by substituting the intermediate state into the two-event state equation.

$$\begin{array}{rl} 1:3,\{V32d\} \rightarrow 1:2\_WDDA,(WDD) & \text{create} \\ & \text{add to known} \\ 1:2\_WDDA,\{WDDA\} \rightarrow 1:2,(No\ action) & \text{substitute in 5.32} \\ & \text{add to known} \end{array} \tag{5.33}$$

The second two-event state equation in our specification list and its possible re-orderings are shown below.

$$\begin{array}{rl} 1:3,\{V32,R23\} \rightarrow 1:3,(Send\ RA3\ no\ data) & \text{spec 5.16} \\ 1:3,\{R23,V32\} \rightarrow 1:3,(Send\ RA3\ no\ data) & \text{reordered from spec} \end{array} \tag{5.34}$$

We then split these multiple-event state equations into single-event state equations using state-equation operators described earlier.

$$\begin{array}{rl} 1:3,\{V32,R23\} \rightarrow 1:3,(Send\ RA3\ no\ data) & \text{spec 5.16} \\ 1:3,\{V32\} \rightarrow 1:2,(No\ action) & \text{known 5.31} \\ 1:2,\{R23\} \rightarrow 1:3,(Send\ RA3\ no\ data) & \text{inferred by substitution} \\ & \text{consistent with 5.14} \end{array} \tag{5.35}$$

$$\begin{array}{rl} 1:3,\{R23,V32\} \rightarrow 1:3,(Send\ RA3\ no\ data) & \text{reordered 5.34} \\ 1:3,\{R23\} \rightarrow 1:3,(Stall) & \text{known 5.24} \\ 1:3,\{V32,R23\} \rightarrow 1:3,(Send\ RA3\ no\ data) & \text{order after stalling} \\ & \text{consistent with 5.16} \end{array} \tag{5.36}$$

The third two-event state equation from the specification list and its re-orderings are,

$$\begin{array}{rl} 1:3,\{V32,V21\} \rightarrow 1:1,(No\ action) & \text{spec 5.17} \\ 1:3,\{V21,V32\} \rightarrow 1:1,(No\ action) & \text{reordered from spec} \end{array} \tag{5.37}$$

Applying state equation reduction operators on them to split the multiple-event state equations into single-event state equations.

$$\begin{aligned}
1:3,\{V32,V21\} &\rightarrow 1:1,(\textit{No action}) && \text{spec 5.17} \\
1:3,\{V32\} &\rightarrow 1:2,(\textit{No action}) && \text{known 5.31} \\
1:2,\{V21\} &\rightarrow 1:1,(\textit{No action}) && \text{inferred by substitution} \\
& && \text{consistent with 5.14}
\end{aligned} \tag{5.38}$$

$$\begin{aligned}
1:3,\{V21,V32\} &\rightarrow 1:1,(\textit{No action}) && \text{reordered 5.37} \\
1:3,\{V21\} &\rightarrow 1:1\_V32,(\textit{No action}) && \text{not known, create} \\
& && \text{add to known} \\
1:1\_V32,\{V32\} &\rightarrow 1:1,(\textit{No action}) && \text{inferred by substitution} \\
& && \text{add to known}
\end{aligned} \tag{5.39}$$

The fourth two-event state equation from the specification list and its re-orderings are,

$$\begin{aligned}
1:3,\{V31,R12\} &\rightarrow 1:2pRA2,(RDD) && \text{spec 5.21} \\
1:3,\{R12,V31\} &\rightarrow 1:2pRA2,(RDD) && \text{reordered from spec}
\end{aligned} \tag{5.40}$$

Applying state equation reduction operators on them to split the multiple-event state equations into single-event state equations.

$$\begin{aligned}
1:3,\{V31,R12\} &\rightarrow 1:2pRA2,(RDD) && \text{spec 5.21} \\
1:3,\{V31\} &\rightarrow 1:1,(\textit{No action}) && \text{known 5.31} \\
1:1,\{R12\} &\rightarrow 1:2pRA2,(RDD) && \text{inferred by substitution} \\
& && \text{consistent with 5.14}
\end{aligned} \tag{5.41}$$

$$\begin{aligned}
1:3,\{R12,V31\} &\rightarrow 1:2pRA2,(RDD) && \text{reordered 5.40} \\
1:3,\{R12\} &\rightarrow 1:3,(\textit{Stall}) && \text{known 5.30} \\
1:3,\{V31,R12\} &\rightarrow 1:2pRA2,(RDD) && \text{order after stalling} \\
& && \text{consistent with 5.21}
\end{aligned} \tag{5.42}$$

Finally, the fifth two-event state equation from the specification list and its re-orderings are,

$$\begin{aligned}
1:3,\{V31,R13\} &\rightarrow 1:3pRA3,(RDD) && \text{spec 5.22} \\
1:3,\{R13,V31\} &\rightarrow 1:3pRA3,(RDD) && \text{reordered from spec}
\end{aligned} \tag{5.43}$$

Applying state equation reduction operators on them to split the multiple-event state equations into single-event state equations.

$$\begin{aligned}
1:3,\{V31,R13\} &\rightarrow 1:3pRA3,(RDD) && \text{spec 5.22}\\
1:3,\{V31\} &\rightarrow 1:1,(No\ action) && \text{known 5.31}\\
1:1,\{R13\} &\rightarrow 1:3pRA3,(RDD) && \text{inferred by substitution}\\
& && \text{consistent with 5.14}
\end{aligned} \tag{5.44}$$

$$\begin{aligned}
1:3,\{R13,V31\} &\rightarrow 1:3pRA3,(RDD) && \text{reordered 5.43}\\
1:3,\{R13\} &\rightarrow 1:3,(Stall) && \text{known 5.30}\\
1:3,\{V31,R13\} &\rightarrow 1:3pRA3,(RDD) && \text{order after stalling}\\
& && \text{consistent with 5.22}
\end{aligned} \tag{5.45}$$

The steps performed in examples above is generalized in algorithm 5 where we apply the substitution, creation and stall operators to reduce a two-event state equation into single-event state equation. When solving the two-event state equation, priority is given to substitution operator where a known single-event state equation is substituted into the multi-event state equation to reduce its cardinality. When substitution is not possible, create operation is used to create a new intermediate state, which can then be added to the set of known state equations. This is done except when the current event is a request in which case it gets stalled to maintain *data-value* invariant.

By applying the same principles, state equations with multiple events can be recursively reduced to single-event state equations as shown. Algorithm 5 is a dynamic programming algorithm with memoization for single-event state equation.

Now that all two-event state equations have been reduced, we proceed with reducing three-event state equations. Algorithm 5 can be applied here as well. In equation 5.46 we show an example of reduction of a state equation with a cardinality of three. It is to be noted that even if this state equation has three coherence events, they are only 2! possible because memory responses will always follow the coherence event that creates the memory request. The three-event state equation and its re-orderings are as follows.

$$\begin{aligned}
1:3,\{V32d,WDDA,V21\} &\rightarrow 1:1,(No\ action) && \text{spec 5.17}\\
1:3,\{V21,V32d,WDDA\} &\rightarrow 1:1,(No\ action) && \text{reordered 5.25}
\end{aligned} \tag{5.46}$$

Algorithm 5 Recursively solving multiple-event state equation to single-event state equation through state equation operators such as substituion, creation and stalling.

```
    Input $S1, \{M1, M2\} \rightarrow S3, (Action1),$ known
    Output known(solved state equations)
 1: cur_stt ← S1
 2: evt_lst ← [M1,M2]
 3: fnl_stt ← S3
 4: acn ← Action1
 5: procedure FILL_STATE_N(known, cur_stt, evt_lst, fnl_stt, acn)
 6:     cur_evt ← evt_list.deque()
 7:     if empty(evt_list) then                          ▷ Single-event state equation
 8:         known ← fill_state_1(known, cur_stt, cur_evt, fnl_stt, acn)    ▷ Alg 4
 9:     else                                          ▷ Multiple-event state equation
10:         int_stt ← None
11:         int_acn ← None
12:         if (cur_stt,cur_evt) in known then                          ▷ Substitute
13:             int_stt ← known[(cur_stt, cur_evt)]["next_state"]
14:             int_acn ← known[(cur_stt, cur_evt)]["action"]
15:             if int_acn is ‘‘Stall’’ then
16:                 return
17:         else                                                    ▷ Stall or create
18:             if evt_type(cur_evt) is ‘‘Request’’ then
19:                 known[(cur_stt, cur_evt)]["next_state"] ← cur_stt
20:                 known[(cur_stt, cur_evt)]["action"] ← ‘‘Stall’’
21:                 return
22:             else
23:                 int_stt ← create_int_stt(evt_lst)                       ▷ Alg 3
24:                 int_acn ← get_evt_acn(cur_evt)         ▷ Perform event-based action
25:             known[(cur_stt, cur_evt)]["next_state"] ← int_stt
26:             known[(cur_stt, cur_evt)]["action"] ← int_acn
27:         cur_stt ← int_stt           ▷ Infer new state equation with one less event
28:         fill_state_n(known, cur_stt, evt_lst, fnl_stt, acn)
29:                                       ▷ Reduce inferred state equation recursively
30:     return(known)
```

These equations are reduced in equations 5.47 and 5.48.

$$\begin{aligned}
1:3,\{V32d,WDDA,V21\} &\rightarrow 1:1,(No\ action) && \text{spec 5.17}\\
1:3,\{V32d\} &\rightarrow 1:2\_WDDA,(WDD) && \text{known 5.33}\\
1:2\_WDDA,\{WDDA,V21\} &\rightarrow 1:1,(No\ action) && \text{inferred by substitution}\\
1:2\_WDDA,\{WDDA\} &\rightarrow 1:2,(No\ action) && \text{known 5.33}\\
1:2,\{V21\} &\rightarrow 1:1,(No\ action) && \text{inferred by substitution}\\
& && \text{consistent with 5.14}
\end{aligned} \tag{5.47}$$

$$\begin{aligned}
1:3,\{V21,V32d,WDDA\} &\rightarrow 1:1,(No\ action) && \text{reordered 5.25}\\
1:3,\{V21\} &\rightarrow 1:1\_V32,(No\ action) && \text{known 5.39}\\
1:1\_V32,\{V32d,WDDA\} &\rightarrow 1:1,(No\ action) && \text{inferred by substituion}\\
1:1\_V32,\{V32d\} &\rightarrow 1:1\_WDDA,(WDD) && \text{not known, create}\\
& && \text{add to known}\\
1:1\_WDDA,\{WDDA\} &\rightarrow 1:1,(No\ action) && \text{inferred by substitution}\\
& && \text{add to known}
\end{aligned} \tag{5.48}$$

Putting it all together,by progressively solving state equations of increasing cardinality from the specification and their re-orderings, we can identify all possible transitions required to generate the DC protocol state machine as shown in algorithm 6. The *sort_by_cardinality*() function in algorithm 6 sorts specification state equation in increasing order of cardinality and within a cardinality, the equations can be solved in any order. The selected state equation and its possible re-orderings are then solved, with the individual state transitions stored to be reused later. Finally we have a list of single-event state equations that can be used to build the DC protocol.

Deadlocks in protocol state machine: Once the protocol state machine is generated, we can represent it in the form of a graph and by checking if the graph is acyclic, we can ensure there are no circular dependencies in a coherence transactions and hence no deadlocks in the protocol state machine.

Algorithm 6 Solve specification state equations one by one in order of increasing cardinality. All possible reorderings of specification state equations are also solved.

```
    Input spec_st_eqns_lst (specification state equations)
    Output known (solved state equations set)
1: procedure SOLVE_SPEC(spec_st_eqns_lst)
2:     known ← {}
3:     for cur_stt,evt_lst,fnl_stt,acn in sort_by_cardinality(spec_st_eqns_lst) do
4:         for evt_lst_order in permute(evt_lst) do                    ▷ reorderings
5:             if possible(evt_lst_order) then
6:                 known ← fill_state_n(known,cur_stt,evt_lst_order,fnl_stt,acn)
7:                                                                      ▷ Alg 5
8:     return (known)
```

5.7 Summary

In this chapter, we considered the interaction between the CPU's LLC and FPGA's DC and limited our scope to DC not issuing forward downgrade transactions. The aim of this chapter was to construct a state machine that allows CPU to coherently access FPGA attached memory. Since we did not have a formal specification of the CPU's protocol, we used the DC protocol model to enumerate *all possible pathways* the CPU can take in its interaction with the DC.

Using these pathways, we were able to identify the coherence transactions that would constitute the specification and represent them in the form of state equations. Then we solved each state equation from the specification along with all of its re-orderings, all the while making design choices that allows coherence invariants to be maintained, conflicts to be handled, no possibility of deadlocks, and for optimization of performance along the critical path.

We then converted these design choices and steps taken to solve these state equations into an algorithm that takes in the state equations from the specification, identifies all possible reordered state equations and solves them to generate a DC protocol state machine in the form of a state table that allows the CPU to coherently access the FPGA address space.

An important point to be noted is that the specification state equations are written by us and hence are modifiable within reason. These modifications can reduce the burden of the protocol at the application level and should be a considered when building an application. A second point is that the formalism developed in this chapter will not change as long as we have a directory-based write-invalidate coherence protocol with no NACKs.

In the next chapter, we build upon the contents of this chapter by allowing the FPGA to initiate forward downgrade coherence transactions.

6
Specifying Coherence Transactions Initiated by DC

6.1 Introduction

In chapter 5, we looked at building a state machine that handles CPU initiated coherence transactions accessing an FPGA homed cache line. We restricted ourselves to CPU initiated coherence transactions and assumed that the DC does not initiate any forward-downgrade transactions. Rule 4 in section 4.3 allows the FPGA to initiate forward-downgrade coherence transactions for FPGA-homed cache lines. In this chapter we will update the specification with more state equations that represents forward-downgrade coherence transactions and how it affects the RS of the cache line in the DC's directory. We will also see how *no* changes have to be made to the algorithms discussed in the previous chapter to accommodate solving these new state equations.

We continue to assume that the FPGA applications cannot issue any caching transactions and the HS of the cache line is I in DC's directory.

The structure of this chapter is as follows.

1. Section 6.2 discusses forward-downgrade transactions.

2. Section 6.3 provides three initial conditions that we want to apply to the DC protocol model to identify forward-downgrade coherence transactions.
3. Sections 6.5, 6.5 and 6.6 shows how forward-downgrade transactions can be specified for the three initial conditions. It also discusses about generating the state machine.

6.2 Forward-Downgrade Transactions

Section 4.4 identifies the coherence messages involved in a forward-downgrade transaction. Depending on the remote state (RS) of the cache line in the DC's directory, the DC can issue three different forward-downgrade requests. If the RS of the cache line is S (step 2), the FPGA can issue an F21 to downgrade the state of the cache line from S to I (step 2 to 1). If the RS of the cache line is E/M (step 3), the FPGA can issue an F32 to request downgrade from E/M to S (step 3 to 2), or issue an F31 to request downgrade from E/M to I (step 3 to 1). The forward-downgrade requests F32 and F31 also causes any dirty data in the CPU's LLC to be written back (cleaned).

Clean and clean-invalidate operations: Based on these coherence messages, we can define two types of operations that can be achieved through forward-downgrade transactions namely, *clean* and *clean-invalidate*. A clean operation on a cache line, upon completion, ensures that the CPU's LLC does not have any dirty copy of the cache line and the FPGA memory has the most up-to-date value. Alternatively, it ensures the state of the cache line is either S or I in the CPU's LLC but never E/M. A clean-invalidate operation, upon completion, ensures that the CPU's LLC does not have any (dirty or clean) copy of the cache line. That is, the state of the cache line in the CPU's LLC is I and the FPGA's memory has the most up-to-date value for the cache line

There can be multiple reasons for the DC to initiate a forward-downgrade coherence transaction for example, to free up internal directory resources, or handling clean or clean-invalidate requests made by applications in the application layer. Irrespective of the cause, when initiating a forward-downgrade transaction, the DC protocol state machine issues a forward-downgrade request for a cache line and transitions to an *intermediate remote state*. This intermediate RS indicates that a forward-downgrade transaction is in progress and that the protocol state machine is waiting for a response.

6.3 Initial Conditions

Similar to initial conditions in section 5.2, We identify the following initial conditions in the DC model, to which we apply the rules of interaction to get all forward-downgrade coherence transactions that are possible. The initial conditions are as follows.

- **DC State: I:I, No forwards issued:** When the RS of the cache line is I in the DC's directory, no forward-downgrade transactions can be issued since the CPU does not have the cache line cached.
- **DC State: I:S, DC issues F21:** When the RS of the cache line is S, the DC can request a downgrade from S to I by issuing an *F21*. The RS of the cache line gets updated to intermediate state *1_A21*. This intermediate state indicates that a forward-downgrade transaction is in progress and that the RS of the cache line would transition from 1_A21 to 1 (I) when coherence message A21 is received.
- **DC State: I:E, DC issues F32:** When the RS of the cache line is E/M, the DC can request a downgrade from E/M to S by issuing an *F32*. The RS of the cache line gets updated to intermediate state *2_A32d* indicating that the cache line would transition from RS 2_A32d to 2 when coherence message *A32d* is received.
- **DC State: I:E, DC issues F31:** When the RS of the cache line is E/M, the DC can request a downgrade from E/M to I state by issuing an *F31*. The RS of the cache line gets updated to intermediate state *1_A31d* indicating that the cache line would transition from RS 1_A31d to 1 when coherence message *A31d* is received.

In the following sections, we apply the rules of interaction to these initial conditions to identify the state equations to be added to the specification.

6.4 DC State: I:S, DC Issues F21

In this initial condition, the DC had the remote state of a cache line as S in its directory when trying to *clean-invalidate* a cache line. The DC issues an F21 request to the CPU, with an intention of downgrading the state of the cache line from S to I in the CPU's LLC. Once F21 is sent, the RS of the cache line gets updated to 1_A21 in the DC's directory.

Although the state of the cache line in the CPU's LLC was S at some point, the state could have changed while the forward-downgrade request was in transit. As such, what the CPU does in response to the forward request depends on the actual state of the cache line in the CPU's LLC when it receives the forward request. From the context of the DC protocol model (section 4.2), lets look at all possible states in which the CPU could receive the forward-downgrade request.

6.4.1 CPU pathways

Figure 6.1 shows all possible states for the cache line in the CPU when the forward-downgrade request F21 can be received. Each scenario is depicted by an encircled alphabet and is described below.

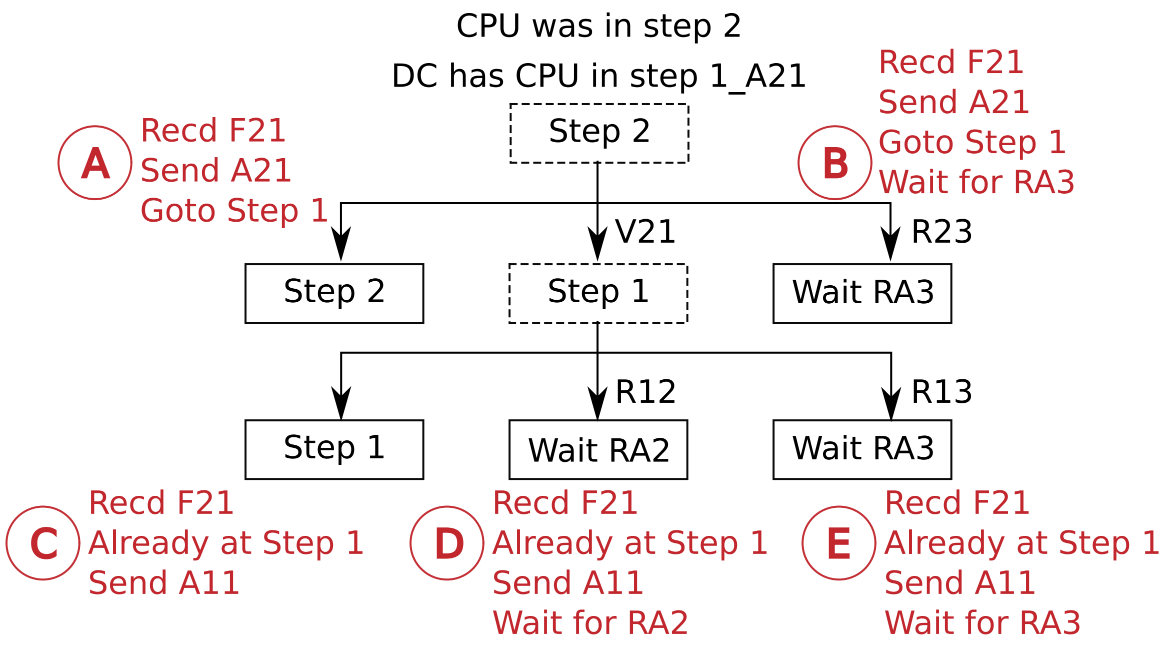


Figure 6.1: *Possible states of cache line in CPU (step) when F21 is received: The cache line can be at step 2, or could have made a request from step 2 to 3, or could have come down to step 1, or could have come down to step 1 and made requests to go to either step 2 or step 3 when F21 was received by the CPU.*

(A) The state of the cache line is S in the CPU's LLC when it receives F21.

(B) The state of the cache line was originally S and the CPU has issued an upgrade request to E (R23) when it receives F21.

Ⓒ The CPU has voluntary downgraded the cache line from S to I state by issuing a V21 and, at this point, it receives F21.

Ⓓ The CPU has voluntary downgraded the cache line from S to I state by issuing a V21 and has then requested an upgrade to S (R12). The CPU receives F21 when waiting for a response to the upgrade request.

Ⓔ The CPU has voluntary downgraded the cache line from S to I state by issuing a V21 and has then requested an upgrade to E (R13). The CPU receives F21 when waiting for a response to the upgrade request.

The reader can convince themselves that based on the DC protocol model in section 4.2, there cannot be any more scenarios given the initial condition. We will now apply the rules of interaction (section 4.4) for each of these scenarios in order to identify the pathways the CPU can take.

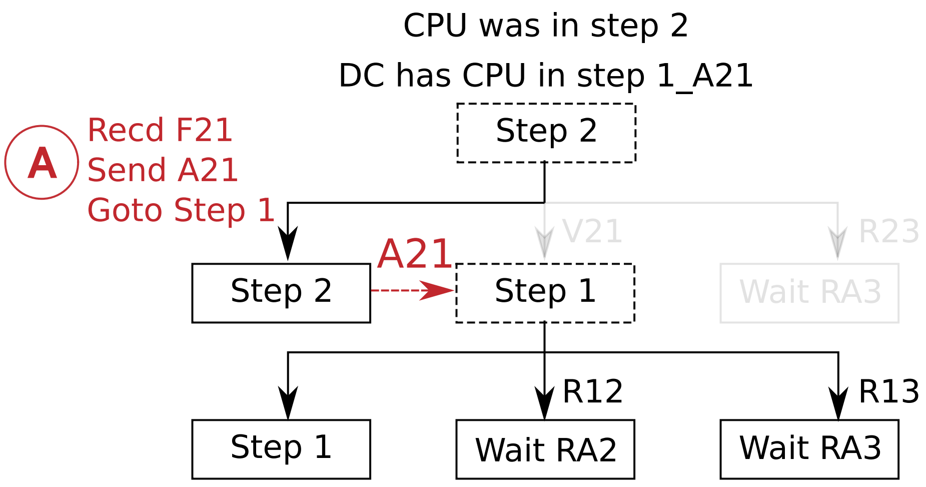


Figure 6.2: *Pathways CPU can take when F21 is received, scenario A: The CPU was at step 2 when it received F21. The CPU sends A21 and comes down to step 1. Then it can remain at step 1 or issue upgrade requests to step 2 or 3 (R12, R23) and wait for a response.*

CPU pathways scenario Ⓐ : In scenario marked Ⓐ in Figure 6.1, the CPU has the cache line in S state in its LLC when it receives F21. There are no conflicts in this scenario and the CPU acknowledges the forward downgrade request with response A21 and downgrades the state of the cache line from S to I in accordance with interaction-rule Rule4. Let us look at what pathways CPU can take from there.

This case is represented in our model with the CPU being in step 2 when it receives F21. The CPU would acknowledge this request with an A21 and come down to step 1. At step 1, it can continue remaining in step 1 or request to go up to step 2 or step 3. Upon issuing an upgrade request, the CPU waits for a response. These pathways are shown in Figure 6.2 and are described below.

(1) The CPU issues A21 to invalidate the cache line and the cache line remains invalid (CPU continues to remain in step 1).

(2) The CPU issues A21 to invalidate the cache line and follows it up with an upgrade request to S (R12). It then waits for RA2, the response to upgrade request.

(3) The CPU issues A21 to invalidate the cache line and follows it up with an upgrade request to E (R13). It then waits for RA3, the response to upgrade request, RA3.

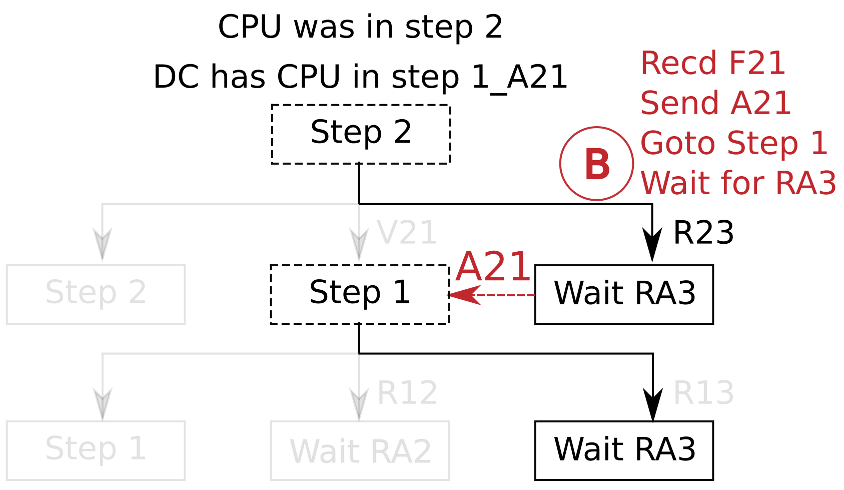


Figure 6.3: *Pathways CPU can take when F21 is received, scenario B: The CPU has made an upgrade request R23 when it receives F21. The CPU comes down to step 1 by issuing A21 and continues to wait for a response to the upgrade request.*

CPU pathways scenario (B) : In scenario (B) in Figure 6.1, the CPU had a S copy of the cache line and has made an upgrade request to E (R23) by the time F21 was received. Since the CPU has not received a response to the upgrade request yet, it is effectively still in S state. The CPU issues an A21 and downgrades to I where it continues to wait for a response to the previously issued upgrade request.

This case is represented in our model where the CPU is in step 2, has issued an upgrade from step 2 to step 3 and is waiting for a response when it receives an F21. In this case, the CPU is effectively in step 2 when it receives the downgrade request and so it acknowledges the request with an A21 and goes down to step 1. At step 1 it cannot make a new upgrade request since it already has an upgrade request (R23) in progress that was issued previously (Rule 5). So the CPU continues to wait for a response (RA3) at step 1. This gives the pathway shown in Figure 6.3 and described below.

(4) The CPU issues A21 to invalidate the cache line and continues to wait for response RA3 for its previously issued upgrade request R23.

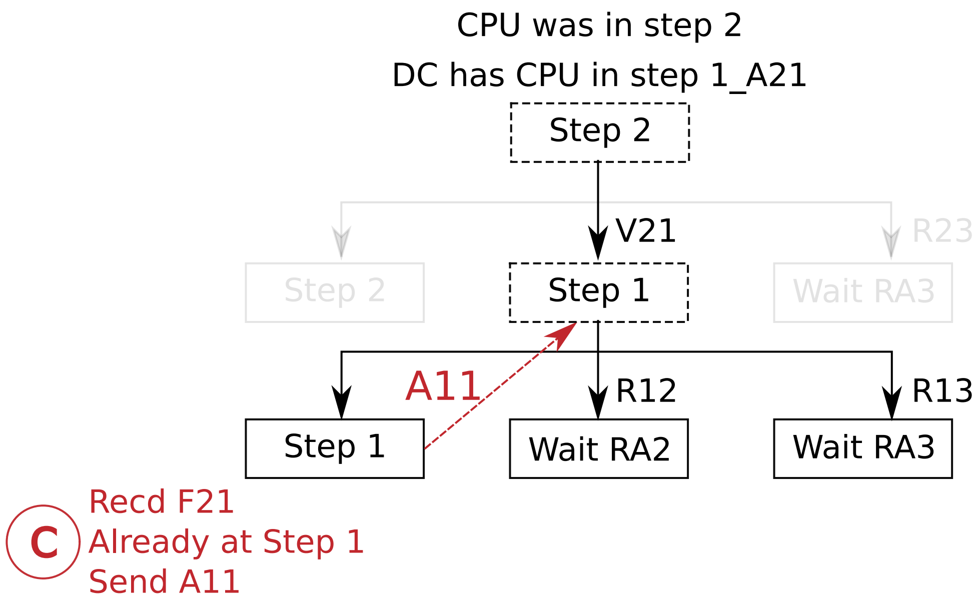


Figure 6.4: *Pathways CPU can take when F21 is received, scenario C: The CPU has already voluntarily come down to step 1 (by issuing V21) when it receives F21. Since the CPU is already at step 1, it issues A11 as a conflict response. The CPU can then continue to remain in step 1 or make upgrade requests from step 1 to 2 (R12) or 3 (R13).*

CPU pathways scenario (C) : In the scenario (C) in Figure 6.1 The CPU had voluntarily downgraded the S cache line to I state by issuing a V21 when it received the forward-downgrade F21. Since the cache line is already invalid, the CPU responds with conflict response A11 as a response to F21 (Figure 4.5). Through A11, the CPU communicates that it has received the forward-downgrade request but has already downgraded at that point and there would be voluntary downgrades in transit.

This case is represented in our model where the CPU has come down to step 1 voluntarily by issuing a V21 and is at step 1 when it receives F21. The CPU issues A11 and then can either continue remaining in step 1 or make upgrade requests from step 1 to step 2 or step 3. Once upgrade requests are issued, the CPU cannot issue any more coherence messages and waits for the response. This gives the pathway shown in Figure 6.4 and described below.

(5) The CPU issues A11 and continues to have I copy.

(6) The CPU issues A11 to indicate the state of the cache line is already I. It then makes an upgrade request (R12) from I to S and waits for response RA2.

(7) The CPU issues A11 to indicate the state of the cache line is already I. It then makes an upgrade request (R13) from I to E and waits for response RA3.

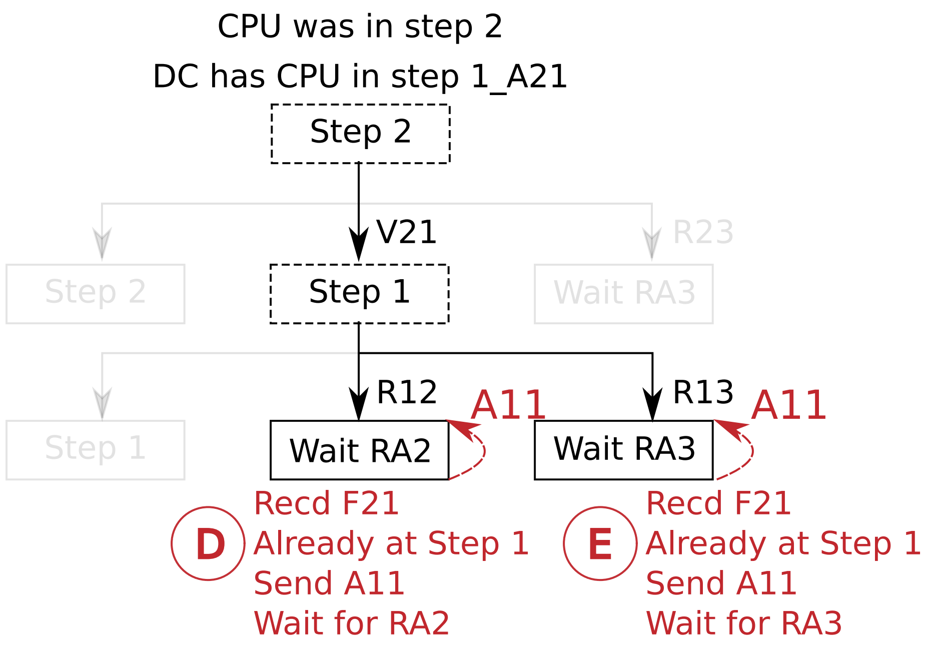


Figure 6.5: *Pathways CPU can take when F21 is received, scenarios D and E: The CPU has voluntarily come down from step 2 to 1 by issuing V21 and has made upgrade requests (R12 or R13). The CPU receives F21 when it is waiting for a response to the upgrade requests. The CPU issues conflict response A11 indicating that it is already at step 1 and continues to wait for a response to the upgrade request.*

CPU pathways scenarios (D) (E): In both these scenarios (Figure 6.1), the CPU had voluntarily downgraded from S to I state by issuing V21. Once downgraded, it has made an upgrade request from I to S ((D)) or from I to E ((E)). The CPU is waiting for a response to these upgrade requests when it receives the forward-downgrade request F21. Since the CPU has not received a response to its upgrade request from invalid state, the effective state of the cache line is still I. The CPU hence responds with conflict response A11 and continues waiting for a response.

In our model, this case is represented by the CPU coming down from step 2 to 1 by issuing voluntary downgrade V21. At step 1, the CPU has issued an upgrade request and is waiting for a response. The CPU responds with an A11 indicating that F21 was received and voluntary downgrades are in transit and continues waiting for a response to the upgrade requests. The CPU cannot issue any more coherence messages since Rule 5 would be violated. These pathways are shown in Figure 6.5 and described below.

(8) The CPU issues A11 and waits for upgrade response RA2.

(9) The CPU issues A11 and waits for upgrade response RA3.

Now we have seen the different ways the CPU can act depending on the state of the cache line in the CPU's LLC when it receives the forward-downgrade request F21 given this initial condition. Next we will specify the coherence transactions in these pathways from the perspective of DC using state equations.

6.4.2 Specification and maintaining coherence invariants

When the DC issues F21, the cache line could be in either S or I state in the CPU's LLC and never E/M. This means the CPU can, at best, have a read-only copy of the cache line and the FPGA memory has the most up-to-date value of the cache line. Since the CPU is the only node that can currently read or write to the cache line, SWMR invariant is trivially maintained.

For all pathways, the initial HS:RS of the cache line in the DC's directory is 1:1_A21 when the DC starts receiving forward-downgrade responses.

For CPU pathway (1), the DC receives forward-downgrade-response A21. Receiving A21 confirms that the CPU has received the forward-downgrade request and has downgraded the state of the cache line from S to I in its LLC as a response. It also guarantees that there are no voluntary downgrades in transit. In this case, the RS of the cache line in

DC's directory can be updated to I. This is represented by state equations in equation 6.1 and can be added to our list of specification state equations.

$$1:1\_A21,\{A21\} \rightarrow 1:1,(No\ action) \quad \text{Handles CPU pathway } \textcircled{1} \tag{6.1}$$

For CPU pathway ② and ③, the DC receives A21 followed by an upgrade request R12 and R13 respectively. The DC has to account for the voluntary downgrade and read the FPGA memory to respond to the upgrade request (RA2 for R12 and RA3 for R13) to maintain coherence invariants. Once the response is sent, the state of the cache line is updated to S (for R12) or E (for R13) depending on the upgrade request. This gives the following state equations shown in equation 6.2 optimized for read critical path.

$$\begin{aligned}1:1\_A21,\{A21,R12\} &\rightarrow 1:2pRA2,(RDD) \quad \text{Handles } \textcircled{2}\\ 1:2pRA2,\{RDDA\} &\rightarrow 1:2,(Send\ RA2)\\ 1:1\_A21,\{A21,R13\} &\rightarrow 1:3pRA3,(RDD) \quad \text{Handles } \textcircled{3}\\ 1:3pRA3,\{RDDA\} &\rightarrow 1:3,(Send\ RA3)\end{aligned} \tag{6.2}$$

For CPU pathway ④, the DC receives an upgrade request R23 followed by forward-downgrade response A21 (order issued by the CPU). The CPU does not have a valid copy of the data and is waiting for exclusive access for this cache line. The DC has to read the FPGA memory and respond to the upgrade request with exclusive access RA3 to maintain coherence invariants. Once RA3 is sent, the RS of the cache line is updated to E in the DC's directory. This is represented by the state equation in equation 6.3.

$$\begin{aligned}1:1\_A21,\{R23,A21\} &\rightarrow 1:3pRA3,(RDD) \quad \text{Handles } \textcircled{4}\\ 1:3pRA3,\{RDDA\} &\rightarrow 1:3,(Send\ RA3)\end{aligned} \tag{6.3}$$

For CPU pathway ⑤, if the DC receives coherence events in the order issued by the CPU, it would receive V21 followed by A11. The DC has to account for both response messages before updating the RS of the DC's directory with I. In pathways ⑥ and ⑦, the CPU issues additional upgrade from I to S or E requests. Since the CPU does not have any copies of the cache line, it is the responsibility of the DC to read the FPGA memory, respond to the requests and update its internal RS. The specification state equations are shown in equation 6.4

$$\begin{aligned}
1:1\_A21, \{V21, A11\} &\rightarrow 1:1, (No\ action) \quad \text{Handles } \textcircled{5}\\
1:1\_A21, \{V21, A11, R12\} &\rightarrow 1:2pRA2, (RDD) \quad \text{Handles } \textcircled{6}\\
1:2pRA2, \{RDDA\} &\rightarrow 1:2, (Send\ RA2)\\
1:1\_A21, \{V21, A11, R13\} &\rightarrow 1:3pRA3, (RDD) \quad \text{Handles } \textcircled{7}\\
1:3pRA3, \{RDDA\} &\rightarrow 1:3, (Send\ RA3)
\end{aligned} \tag{6.4}$$

In CPU Pathways ⑧ and ⑨, the order of issuing coherence messages by CPU is V21, followed by R12, and then A11. The DC has to account for all these responses and then read the FPGA memory to respond to the upgrade requests. This is shown in specification state equations in equation 6.5

$$\begin{aligned}
1:1\_A21, \{V21, R12, A11\} &\rightarrow 1:2pRA2, (RDD) \quad \text{Handles } \textcircled{8}\\
1:2pRA2, \{RDDA\} &\rightarrow 1:2, (Send\ RA2)\\
1:1\_A21, \{V21, R13, A11\} &\rightarrow 1:3pRA3, (RDD) \quad \text{Handles } \textcircled{9}\\
1:3pRA3, \{RDDA\} &\rightarrow 1:3, (Send\ RA3)
\end{aligned} \tag{6.5}$$

Thus we have got specification state equations on how the DC should handle all possible CPU pathways given the initial conditions.

6.4.3 Building the state machine

The design choices and algorithm described in chapter 5 can be used without any modifications to solve the additional specification state equations and its reorderings introduced in this section. Once all state equations are solved, the protocol state machine can be generated and checked for deadlocks. The state machine will be optimized for performance along the read critical path since that optimization is baked into the specification.

6.5 DC State: I:E, DC Issues F32

In this initial condition, the DC had the remote state of a cache line as E in its directory when trying to *clean* the cache line. The DC issues an F32 forward request to the CPU, with an intention of downgrading the state of the cache line from E to S in the CPU's LLC. Upon issuing the request, the RS in the DC's directory is updated to 2_A32d to indicate that this forward-downgrade transaction is in progress.

Although the state of the cache line in the CPU's LLC was E, the state could have changed while the forward-downgrade request was in transit. As a result, what the CPU does in response to the forward request depends on the state of the cache line in the CPU's LLC when it receives the request. From the context of the model (section 4.2) lets look at all possible states in which the CPU could receive the forward-downgrade request.

6.5.1 CPU pathways

Figure 6.6 shows all possible states for the cache line in CPU's LLC when the forward-downgrade request F32 could be received. Each scenario is depicted by an encircled alphabet and is described below.

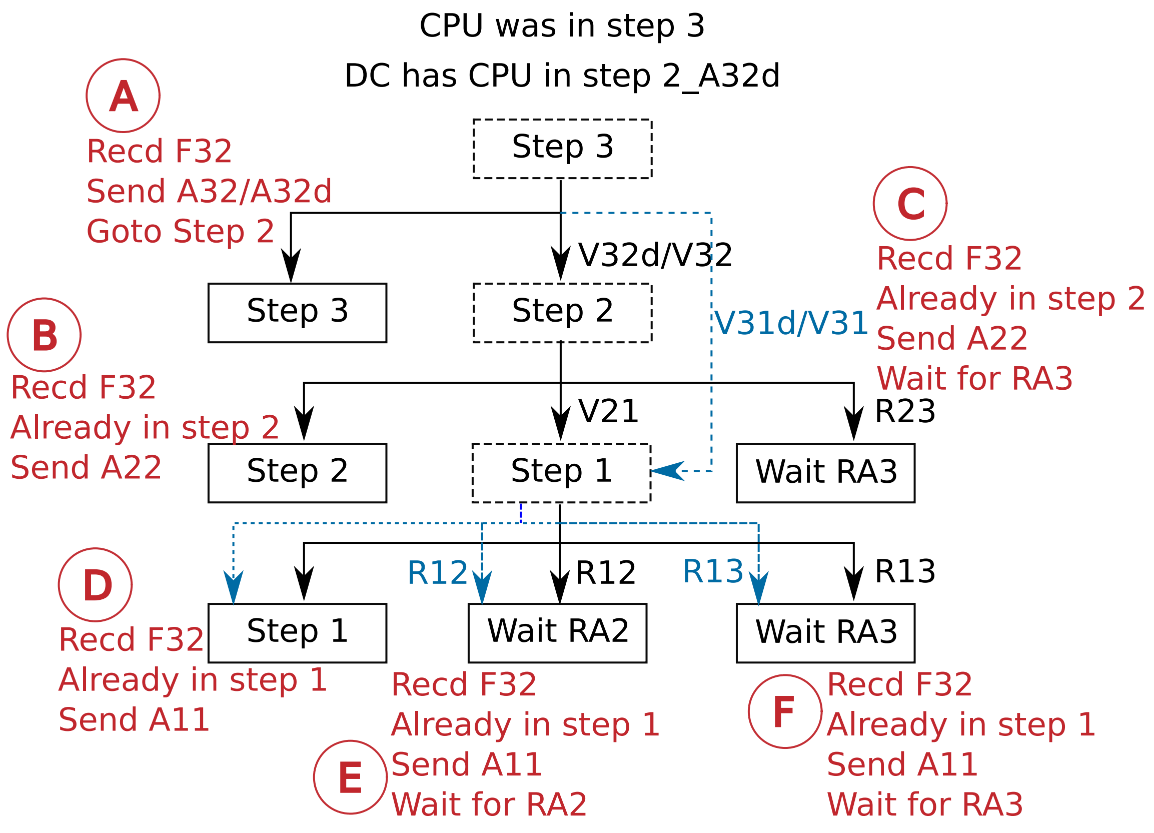


Figure 6.6: *Possible states of CL in CPU when F32 is received: The CPU could have been at step 3, or come down to step 2, or come down to step 2 and made an upgrade request to step 3, or come down from step 2 to 1, or could have come down from step 3 to 1 where it could remain at step 1 or make upgrade requests to step 2 or 3 and wait for response when it received F32.*

Ⓐ The state of the cache line is E in the CPU's LLC when it receives F32.

Ⓑ The state of the cache line was originally E and the CPU had issues a voluntary downgrade (V32 or V32d with dirty data) and downgraded to S state. At this state F32 is received by the CPU.

Ⓒ The state of the cache line was originally E and the CPU had issued a voluntary downgrade (V32 or V32d with dirty data) and downgraded to S state. The CPU has then made an upgrade request from S to E state (R23). The F32 requests is received at this point when the CPU is waiting for a response (RA3) from the FPGA.

Ⓓ The state of the cache line was originally E and the CPU had downgraded this cache line to I state, at which point, the CPU receives the F32. The transition of cache line from E to I state in CPU's LLC could have happened in two ways, the CPU could have downgraded from E to S (by issuing a V32/V32d) and then downgraded from S to I (by issuing a V21), or the CPU could have directly downgraded from E to I by issuing a V31 or V31d (shown by blue-dotted transition line in Figure 6.6). The state of the cache line is I when it receives F32.

Ⓔ This scenario is the same as scenario Ⓓ in the fact that the CPU has downgraded the state of the cache line from E to I. In addition, the CPU has made an upgrade request from I to S state by issuing R12. F32 is received by the CPU when it is waiting for a response (RA2) for the upgrade request.

Ⓕ In this scenario as well, the CPU has downgraded the state of the cache line from E to I like in scenario Ⓓ. In addition the CPU has made an upgrade request from I to E state by issuing R13. F32 is received by the CPU when it is waiting for a response (RA3) for the upgrade request.

Given the DC protocol model section 4.2 and initial condition, there cannot be any other scenario where an F32 is received by the CPU. Lets apply the rules of interaction to each of these scenarios and identify all pathways the CPU can take.

CPU pathways scenario Ⓐ : For scenario Ⓐ in Figure 6.6 the CPU has the cache line in E state when it receives F32. The CPU acknowledges the forward-downgrade request with response A31 (or A31d for dirty data) and downgrades the state of the cache line to S.

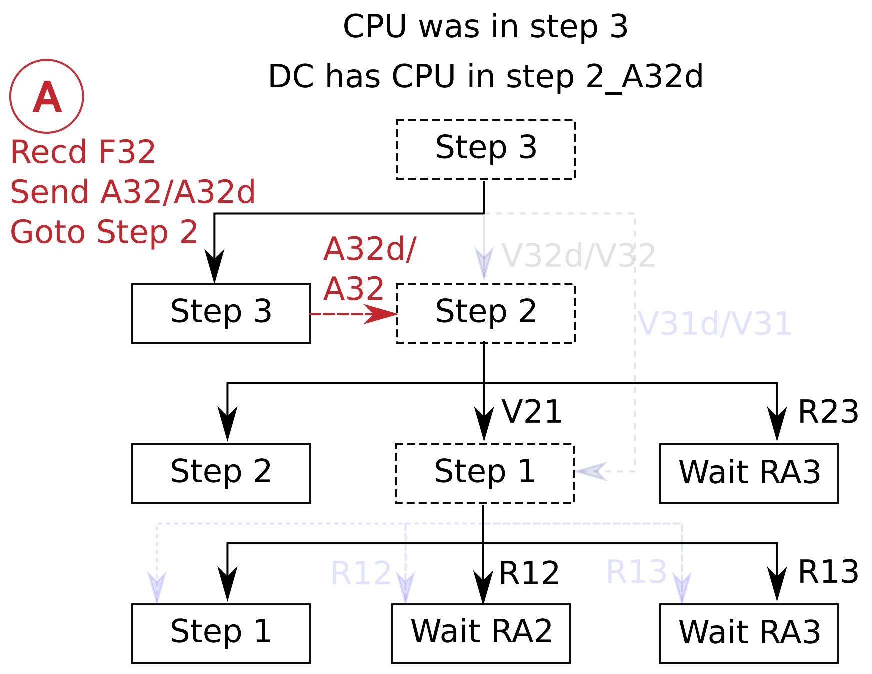


Figure 6.7: *Pathways CPU can take when F32 is received, scenario A: The CPU was in step 3 when it received F32. The CPU issues A32 (or A32d) if dirty and comes down to step 2. Then the CPU can remain at step 2, make an upgrade request to go from step 2 to 3 (R23) or come down to step 1 (V21). At step 1, the CPU can continue to remain there or make upgrade requests to steps 2 (R12) or 3 (R13)*

In the DC protocol model, this is represented by the CPU stepping down from step 3 to step 2 as a response to F32. The CPU can then continue remaining at step 2, or make a request to go up to step 3 and wait for a response. From step 2, the CPU can also choose to come down the stairs to step 1. The CPU can then continue remaining in step 1 or make requests to go up the stairs to either step 2 or step 3 and wait for a response. These pathways are shown in Figure 6.7 and are summarized below with the coherence messages the CPU issues in order.

1. The CPU issues A32/A32d as a response to the F32 and downgrades the cache line to S state, it does not send any more coherence messages for this cache line.
2. The CPU issues A32/A32d to downgrade the cache line to S and follows it up with R23 to request an upgrade from S to E.
3. The CPU issues A32/A32d to downgrade the cache line to S and then voluntarily downgrades to I state by issuing a V21.

(4) The CPU issues A32/A32d to downgrade the cache line to S, then voluntarily downgrades to I state by issuing a V21 and then requests an upgrade from I to S state by issuing an R12.

(5) The CPU issues A32/A32d to downgrade the cache line to S, then voluntarily downgrades to I state by issuing a V21 and then requests an upgrade from I to E state by issuing an R13.

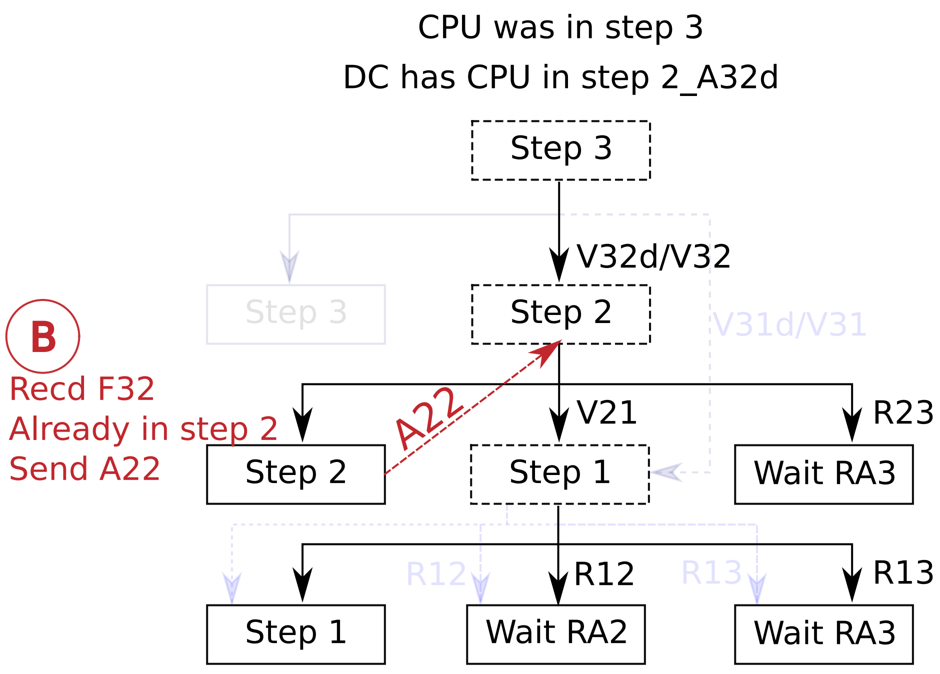


Figure 6.8: *Pathways CPU can take when F32 is received, scenario B: The CPU had already come down from step 3 to 2 by issuing V32 (or V32d if dirty) when it receives F32. The CPU issues conflict response A22 and then can continue to remain at step 2 or come down to step 1 (V21). At step 1, it can remain there or make upgrade requests to step 2 (R12) or step 3 (R13).*

CPU pathways scenario (B) : In this scenario, the CPU has already downgraded the cache line from E to S when it receives the F32. The CPU acknowledges the F32 with conflict response A22 indicating that the cache line is already in S state and that conflict responses are in transit.

This case is represented in the model where the CPU has already voluntarily stepped down from step 3 to step 2 by issuing either V32 or V32d when it receives F32. The CPU responds with A22 and can continue remaining in step 2 or make a request to go to step 3 by issuing an R23. Alternatively, the CPU, after issuing A22, can choose to voluntarily come down to step 1 by issuing a V21. At step 1, the CPU has three choices: remain at step 1, or request to go to step 2 by issuing R12, or request to go to step 3 by issuing R13. These pathways are shown in Figure 6.8 and described below.

(6) The CPU has already issued V32 or V32d when it receives the F32. The CPU issues A22 as a response to F32 and does not issue any more coherence messages for this cache line.

(7) The CPU has V32 or V32d in transit when it issues A22. After issuing A22, the CPU issues an R23 and waits for a response.

(8) The CPU has V32 or V32d in transit when it issues A22. After issuing A22, it voluntarily invalidates this cache line by issuing a V21.

(9) The CPU has V32 or V32d in transit when it issues A22. After issuing A22. The CPU then voluntarily invalidates the cache line by issuing a V21. The CPU then makes a request to upgrade the cache line to S state by issuing R12 and waiting for a response.

(10) The CPU has V32 or V32d in transit when it issues A22. After issuing A22. The CPU then voluntarily invalidates the cache line by issuing a V21. The CPU then makes a request to upgrade the cache line to E state by issuing R13 and waiting for a response.

CPU pathways scenario (C) : In this scenario, the CPU has downgraded the state of the cache line from E to S and then has issued an upgrade request to E when it receives F32. Since the effective state of the cache line is still S, the CPU responds with an A22 and continues waiting for a response to the upgrade request.

In the DC protocol model, the CPU was initially in step 3 and has come down to step 2 by issuing a V32 or V32d. At step 2, it has then requested to go up to step 3 by issuing an R23 when it receives F32. Since the CPU is waiting for a response to the upgrade request, it cannot issue any further coherence events other than response to the forward-downgrade request. For example Rule 5 explicitly forbids the CPU to voluntarily come down to step 1 when the upgrade request from step 2 to 3 is still pending. Thus the only pathway is shown in Figure 6.9 and described below.

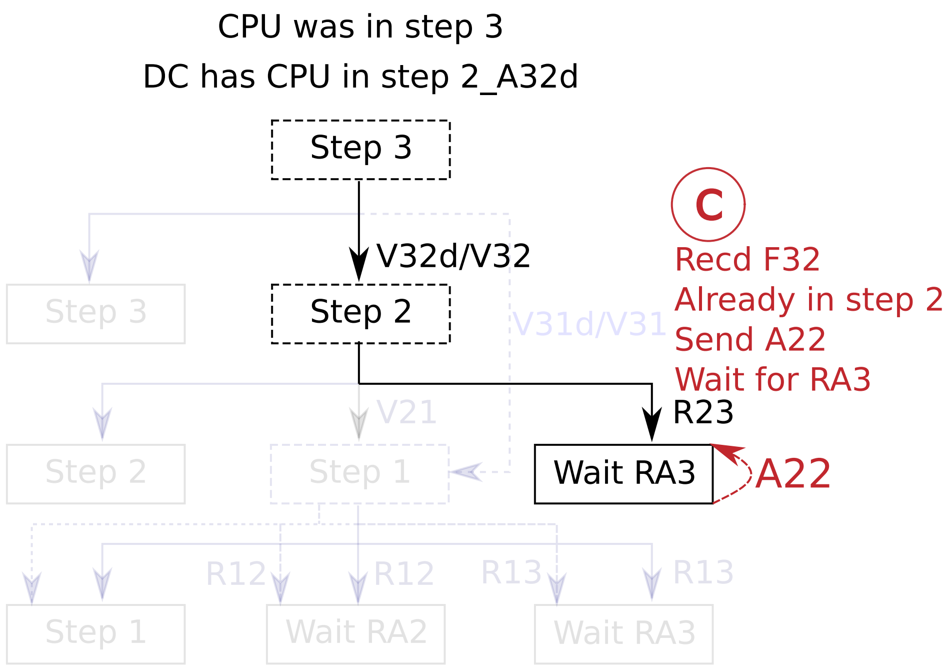


Figure 6.9: *Pathways CPU can take when F32 is received, scenario C: The CPU has come down from step 3 to 2 by issuing V32 (or V32d if dirty) and has made upgrade request from step 2 to 3 (R23) when it receives F32. Since CPU is already in step 2, it sends conflict response A22 and continues to wait for a response to the upgrade request.*

(11) The CPU has V32 or V32d followed by R23 in transit when it receives F32. The CPU responds with A22 and continues waiting for response to the R23 request.

CPU pathways scenario (D) : In this scenario, the CPU has voluntarily downgraded the cache line from E to I when it receives F32. Since the cache line is invalid when the forward-downgrade request is received, the CPU responds with A11 indicating that voluntary downgrades are in transit.

This case is represented in our model where the CPU was initially in step 3 and has voluntarily come down to step 1. The CPU could have done this in two ways: Come down from step 3 to 2 and then from step 2 to 1 by issuing a V32 or V32d followed by V21, or come down directly from step 3 to 1 by issuing V31 or V31d. At this point the CPU receives the forward-downgrade request to come from step 3 to 2, but since the CPU is already in step 1, it responds with a conflict response A11 indicating some voluntary downgrades might be in transit. Once the CPU has issued A11, it can continue to remain at step 1 or make a request to go to step 2 or step 3. The pathways are shown in Figure 6.10

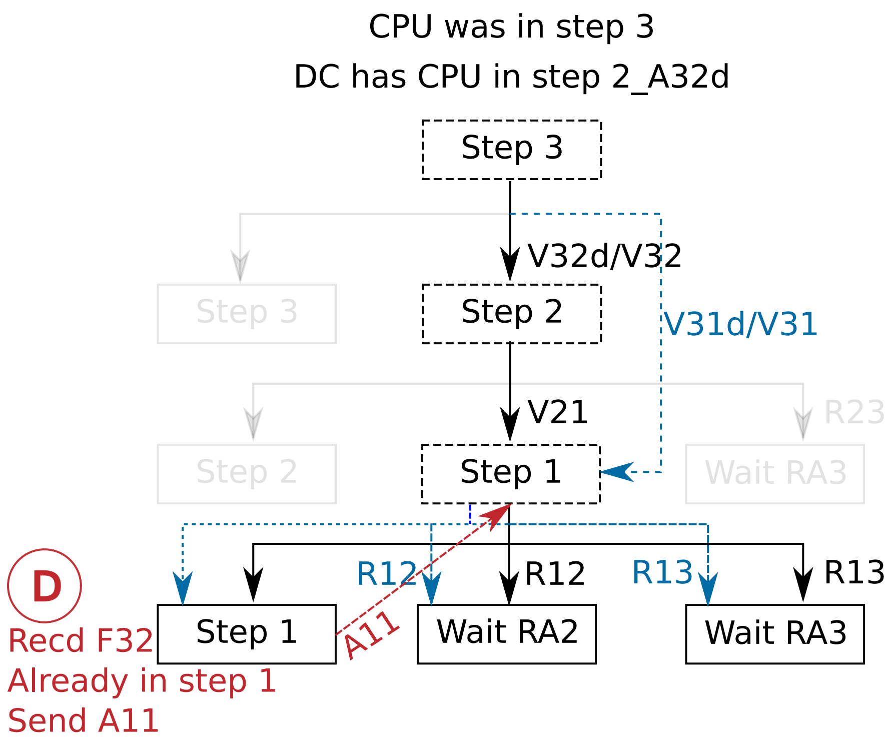


Figure 6.10: *Pathways CPU can take when F32 is received, scenario D: The CPU had voluntarily downgraded to step 3 to step 1 either directly (V31d/V31) or through step 2 (V32d/V32 followed by V21) when it receives F32. Since CPU is in step 1, it responds with conflict response A11 and then can continue remaining in step 1 or make upgrade requests to step 2 (R12) or step 3 (R13).*

and the order of coherence messages issued by the CPU is summarized below.

(12) The CPU has voluntarily downgraded from E to I by issuing a V32 or V32d followed by a V21. The CPU receives F32 at this point and responds with A11. It does not send any other coherence messages for this cache line.

(13) The CPU has voluntarily downgraded from E to I by issuing a V32 or V32d followed by a V21. The CPU receives F32 at this point and responds with A11. The CPU then issues an upgrade request R12 to upgrade the cache line from I to S state.

(14) The CPU has voluntarily downgraded from E to I by issuing a V32 or V32d followed by a V21. The CPU receives F32 at this point and responds with A11. The CPU then issues an upgrade request R13 to upgrade the cache line from I to E state.

(15) The CPU alternatively downgrades from E to I by issuing a V31 or V31d (blue-dotted lines in Figure 6.10) and then receives F32. The CPU responds with A11 and does not issue any other coherence messages for this cache line.

(16) The CPU downgrades from E to I by issuing a V31 or V31d. It receives F32 at this point and responds with an A11. The CPU then issues R12 upgrade request to get the cache line in S state.

(17) The CPU downgrades from E to I by issuing a V31 or V31d. It receives F32 at this point and responds with an A11. The CPU then issues R13 upgrade request to get the cache line in E state.

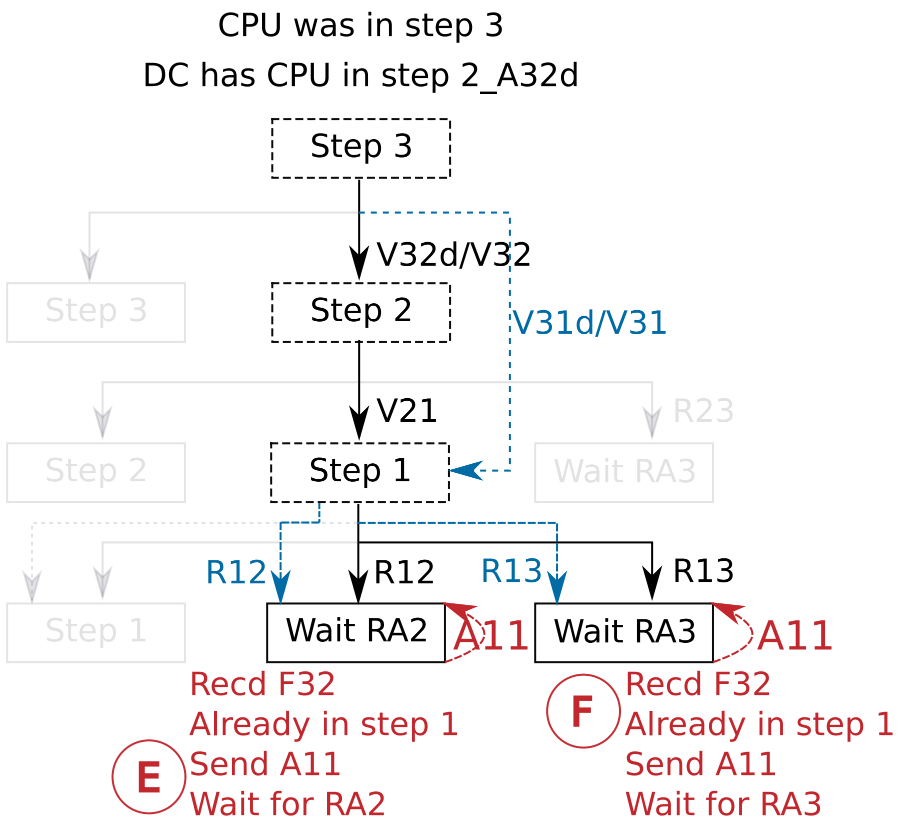


Figure 6.11: *Pathways CPU can take when F32 is received, scenario E, F: The CPU has come down from step 3 to 1 either directly (V31d/V31) or through step 2 (V32d/V32 followed by V21) and has made upgrade request either to step 2 (R12) or step 3 (R13) when it receives F32. The CPU responds with conflict response A11 and continues waiting for a response to the upgrade request.*

CPU pathways scenario (E), (F) : In both these scenarios, the CPU has voluntarily downgraded the cache line from E to I state. The CPU then makes an upgrade request to S ((E)) or E ((F)) and receives F32 when waiting for a response. Since the CPU has not received a response to the upgrade request, the cache line is effectively in I state and the CPU responds with the conflict response A11. The CPU continues to wait for a response to the upgrade request in both the scenarios and cannot issue any other coherence messages.

This is represented in the protocol model with the CPU coming down from step 3 to step 1 by either issuing a V32/V32d followed by V21 or by issuing V31d. The CPU then requests to go up the stairs to step 2 or 3 when it receives F32. The CPU responds with A11 and continues waiting for a response to the upgrade request. These pathways are shown in Figure 6.11 and described below.

(18) The CPU has issued V32 or V32d, followed by a V21, followed by an R12 when it receives F32. The CPU responds with A11 and continues waiting for a response for request R12.

(19) The CPU has issued V32 or V32d, followed by a V21, followed by an R13 when it receives F32. The CPU responds with A11 and continues waiting for a response for request R13.

(20) The CPU has issued V31 or V31d, followed by an R12 when it receives F32. The CPU responds with A11 and continues waiting for a response for request R12.

(21) The CPU has issued V31 or V31d, followed by an R13 when it receives F32. The CPU responds with A11 and continues waiting for a response for request R13.

Now we have identified all pathways the CPU can take given the initial conditions, we can define the specification state equations and check how coherence invariants can be maintained.

6.5.2 Specification and maintaining coherence invariants

The DC should have provided exclusive access for the cache line to the CPU in order for it to have its RS marked as E in its directory. Thus when the DC issues the forward-downgrade, the CPU has the most up-to-date copy of the cache line and the contents of the FPGA would be stale. In section 5.1 we discussed how SWMR invariant is always maintained in our system. To maintain *data-value* invariant, any upgrade requests that arrive before the dirty data is written to memory would have to be stalled. The design choice (design-choice 5.2 to stall all upgrade requests till *all* inferred responses in transit are handled ensures that *data-value* invariant will be maintained. The specifications for the pathways are listed in equations 6.6 and 6.7. The reader can compare the state equations with the pathways to check if they are correct.

$$
\begin{array}{rc}
 & \text{pathway} \\
1:2\_A32d,\{A32\}\rightarrow 1:2,(No\ action) & \textcircled{1} \\
1:2\_A32d,\{A32d,WDDA\}\rightarrow 1:2,(No\ action) & \textcircled{1} \\
1:2\_A32d,\{A32,R23\}\rightarrow 1:3,(Send\ RA3\ no\ data) & \textcircled{2} \\
1:2\_A32d,\{A32d,WDDA,R23\}\rightarrow 1:3,(Send\ RA3\ no\ data) & \textcircled{2} \\
1:2\_A32d,\{A32,V21\}\rightarrow 1:1,(No\ action) & \textcircled{3} \\
1:2\_A32d,\{A32d,WDDA,V21\}\rightarrow 1:1,(No\ action) & \textcircled{3} \\
1:2\_A32d,\{A32,V21,R12\}\rightarrow 1:2pRA2,(RDD) & \textcircled{4} \\
1:2pRA2,\{RDDA\}\rightarrow 1:2(Send\ RA2) & \\
1:2\_A32d,\{A32d,WDDA,V21,R12\}\rightarrow 1:2pRA2,(RDD) & \textcircled{4} \\
1:2\_A32d,\{A32,V21,R13\}\rightarrow 1:3pRA3,(RDD) & \textcircled{5} \\
1:3pRA3,\{RDDA\}\rightarrow 1:3(Send\ RA3) & \\
1:2\_A32d,\{A32d,WDDA,V21,R13\}\rightarrow 1:3pRA3,(RDD) & \textcircled{5} \\
1:2\_A32d,\{V32,A22\}\rightarrow 1:2,(No\ action) & \textcircled{6} \\
1:2\_A32d,\{V32d,WDDA,A22\}\rightarrow 1:2,(No\ action) & \textcircled{6} \\
1:2\_A32d,\{V32,R23,A22\}\rightarrow 1:3,(Send\ RA3\ no\ data) & \textcircled{7} \\
1:2\_A32d,\{V32d,WDDA,R23,A22\}\rightarrow 1:3,(Send\ RA3\ no\ data) & \textcircled{7} \\
1:2\_A32d,\{V32,A22,V21\}\rightarrow 1:1(No\ action) & \textcircled{8} \\
1:2\_A32d,\{V32d,WDDA,A22,V21\}\rightarrow 1:1(No\ action) & \textcircled{8} \\
1:2\_A32d,\{V32,A22,V21,R12\}\rightarrow 1:2pRA2(RDD) & \textcircled{9} \\
1:2\_A32d,\{V32d,WDDA,A22,V21,R12\}\rightarrow 1:2pRA2(RDD) & \textcircled{9} \\
1:2\_A32d,\{V32,A22,V21,R13\}\rightarrow 1:3pRA3(RDD) & \textcircled{10} \\
1:2\_A32d,\{V32d,WDDA,A22,V21,R13\}\rightarrow 1:3pRA3(RDD) & \textcircled{10} \\
1:2\_A32d,\{V32,R23,A22\}\rightarrow 1:3(Send\ RA3\ no\ data) & \textcircled{11} \\
1:2\_A32d,\{V32d,WDDA,R23,A22\}\rightarrow 1:3(Send\ RA3\ no\ data) & \textcircled{11}
\end{array}
\tag{6.6}
$$

$$
\begin{array}{rc}
 & \text{pathway} \\
1:2\_A32d, \{V32, V21, A11\} \rightarrow 1:1(No\ action) & \textcircled{12} \\
1:2\_A32d, \{V32d, WDDA, V21, A11\} \rightarrow 1:1(No\ action) & \textcircled{12} \\
1:2\_A32d, \{V32, V21, A11, R12\} \rightarrow 1:2pRA2(RDD) & \textcircled{13} \\
1:2\_A32d, \{V32d, WDDA, V21, A11, R12\} \rightarrow 1:2pRA2(RDD) & \textcircled{13} \\
1:2\_A32d, \{V32, V21, A11, R13\} \rightarrow 1:3pRA3(RDD) & \textcircled{14} \\
1:2\_A32d, \{V32d, WDDA, V21, A11, R13\} \rightarrow 1:3pRA3(RDD) & \textcircled{14} \\
1:2\_A32d, \{V31, A11\} \rightarrow 1:1(No\ action) & \textcircled{15} \\
1:2\_A32d, \{V31d, WDDA, A11\} \rightarrow 1:1(No\ action) & \textcircled{15} \\
1:2\_A32d, \{V31, A11, R12\} \rightarrow 1:2pRA2(RDD) & \textcircled{16} \\
1:2\_A32d, \{V31d, WDDA, A11, R12\} \rightarrow 1:2pRA2(RDD) & \textcircled{16} \\
1:2\_A32d, \{V31, A11, R13\} \rightarrow 1:3pRA3(RDD) & \textcircled{17} \\
1:2\_A32d, \{V31d, WDDA, A11, R13\} \rightarrow 1:3pRA3(RDD) & \textcircled{17} \\
1:2\_A32d, \{V32, V21, R12, A11\} \rightarrow 1:2pRA2(RDD) & \textcircled{18} \\
1:2\_A32d, \{V32d, WDDA, V21, R12, A11\} \rightarrow 1:2pRA2(RDD) & \textcircled{18} \\
1:2\_A32d, \{V32, V21, R13, A11\} \rightarrow 1:3pRA3(RDD) & \textcircled{19} \\
1:2\_A32d, \{V32d, WDDA, V21, R13, A11\} \rightarrow 1:3pRA3(RDD) & \textcircled{19} \\
1:2\_A32d, \{V31, R12, A11\} \rightarrow 1:2pRA2(RDD) & \textcircled{20} \\
1:2\_A32d, \{V31d, WDDA, R12, A11\} \rightarrow 1:2pRA2(RDD) & \textcircled{20} \\
1:2\_A32d, \{V31, R13, A11\} \rightarrow 1:3pRA3(RDD) & \textcircled{21} \\
1:2\_A32d, \{V31d, WDDA, R13, A11\} \rightarrow 1:3pRA3(RDD) & \textcircled{21}
\end{array}
\tag{6.7}
$$

6.5.3 Building the state machine

The design choices and algorithm described in chapter 5 can be used to solve the additional specification state equations and its reorderings introduced in this section.

6.6 DC State: I:E, DC Issues F31

In this section, the DC had given exclusive access for a cache line to the CPU. The DC then wants to downgrade the state of this cache line from E to I in the CPU's LLC. The DC does this by issuing a forward-downgrade exclusive to invalid (F31) and transitioning the RS of the cache line to intermediate state 1_A31d in its directory. The DC then waits for a response to the forward-downgrade request.

As in the previous cases, the state of the cache line in the CPU's LLC could have changed while F31 is in transit and how the CPU responds to F31 depends on the actual state of the cache line in the CPU's LLC when it receives the request. From the context of our model lets look at all possible states in which the CPU could receive the forward downgrade request.

6.6.1 CPU pathways and specification

The CPU can receive the forward-downgrade request F31 in any of these situations. The pathways taken by the CPU are described in Figure 6.12. No changes have to be made to algorithms in chapter 5 to solve these incorporate these additional state equations.

- The pathways for each scenario Ⓐ is shown in Figure A.1. Please note the specification for each pathway is also shown.
- Pathways for scenario Ⓑ is shown in Figure A.2 along with the specification state equations.
- Pathways for scenario Ⓒ is shown in Figure A.3 along with the specification state equations.
- Pathways for scenario Ⓓ is shown in Figure A.4 along with the specification state equations.
- Pathways for scenarios Ⓔ, Ⓕ is shown in Figure A.5 along with the specification state equations.

The design choices made previously will guarantee the coherence invariants for all the specification state equations and its reorderings and no additional changes are required for the algorithm of the state space exploration tool.

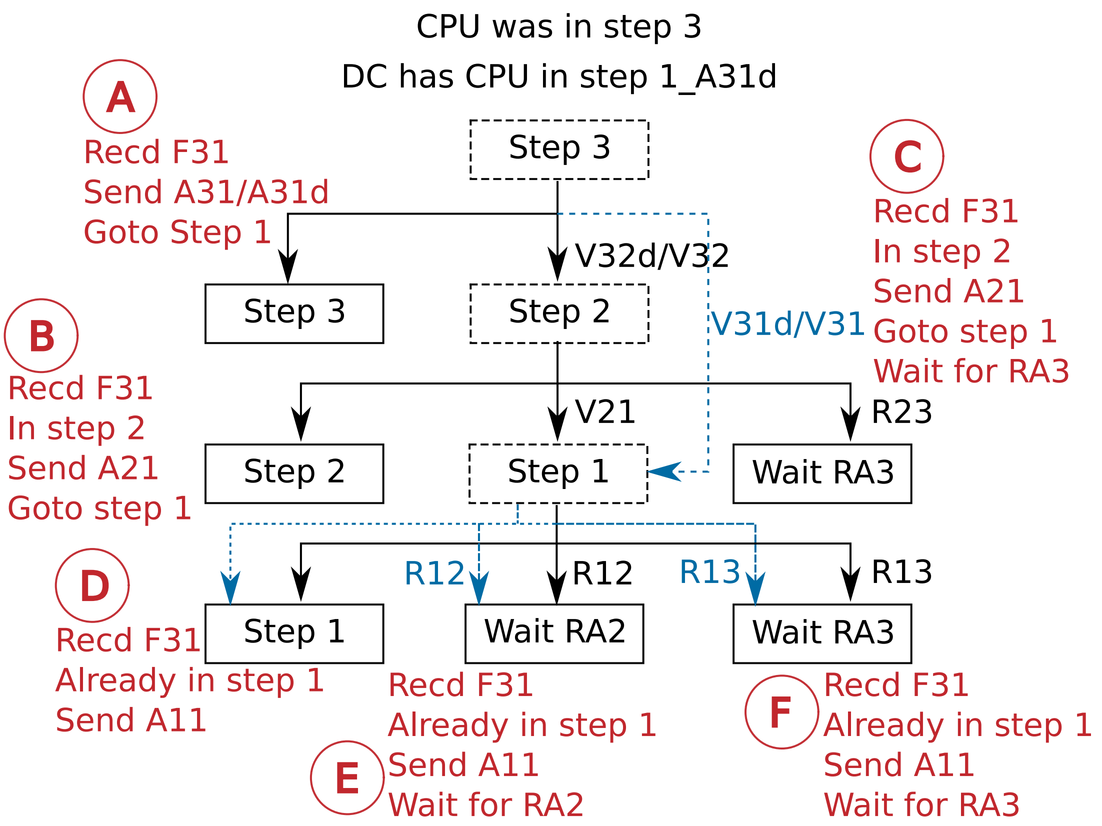


Figure 6.12: *Possible states of CL in CPU when F31 is received: The CPU could have received F31 at step 3, or it could have come down to step 2, or it could have come down to step 2 and made an upgrade request to step 3, or it could have come down to step 1 where it could have made upgrade requests as well.*

6.7 Summary

In this chapter, we identified the different forward downgrade transactions that are possible between the CPU and FPGA. We specifically looked at how these transactions affect the RS of a cache line in DC's directory. We specified the forward downgrade transactions using state equations, which can be fed to the state space exploration tool to generate the state machine.

With each state equation in the specification, we have considered conflicts arising due to reordering by the interconnect, how coherence invariants can be maintained, how deadlocks can be avoided and how performance is optimized at the read and write critical paths.

In this chapter, we limited our discussion to how the RS of the cache line in the DC's directory would change once forward-downgrade transactions are initiated. In the next chapter we will look at what can cause the DC to initiate forward-downgrade transactions. Specifically we will look at how applications on the FPGA interact with the DC.

7 Specifying Coherence Transactions Initiated by Applications

7.1 Introduction

In the three layer model (Figure 3.3), the application (also known as user logic) layer on the FPGA is built on top of the DC protocol layer and relies on the guarantees provided by the protocol and interconnect layers below it. In this chapter we ask the question: *What interface should the DC expose to applications so as to provide useful functionality while abstracting the inner workings of the coherence protocol?*

The lowermost interconnect layer allows for deadlock free exchange of coherence messages. It is optimized for performance and guarantees delivery of messages but does provide any ordering guarantees. Next, The DC protocol layer treats every cache line as independent and maintains coherence invariants at the granularity of a cache line. This layer is deadlock free, optimized for performance and also accounts for reordering of coherence messages by the interconnect layer. Finally the application layer can be built on top of the DC protocol layer and can be used to make interesting associations between unrelated cache lines to achieve certain objectives.

These associations can be in the form of invariants that would have to be maintained across different cache lines and address spaces. An example of such an invariant is guaranteeing that when a specific cache line is in E/M state in CPU's LLC, another (unrelated) cache line is always in I state in CPU's cache. Separating the application layer from the protocol layer allows the FPGA developer to implement different applications without needing to change the protocol layer.

Takeaway 7.1. *Traditionally, coherence protocols aim only to maintain coherence invariants across multiple copies of a single cache line. As a result, applications always interact with the protocol layer through a cache. Not having a cache on the FPGA, allows us to provide a flexible interface between the protocol layer and application layer. This interface can be used by applications on the FPGA to extend the notion of coherence beyond copies of a single cache line to associate multiple unrelated cache lines and address spaces.*

Advantage of the FPGA application layer: The main aim is to target applications on heterogeneous systems where the application has a software component running on the CPU cores and user logic running on the FPGA (FPGA application). The operations performed by memory hierarchy with respect to maintaining coherence is not visible directly to software running on the CPU: software interacts with the CPU's LLC through load, store, clean and invalidate instructions. The aim of the FPGA application is to *transparently* extend the notion of coherence to software on the CPU. For example, application specific coherence requirements can be offloaded to the FPGA application and, can be taken for granted by the software component (see materialized view application in subsection 9.6.3 for example).

Local coherence transactions: The DC protocol layer allows for interaction with the application layer through *local* coherence transactions. In this chapter, we focus on local transactions that applications on the FPGA can initiate with the DC protocol state machine. The DC provides a simplified interface for applications on the FPGA to interact with the coherence protocol, *specifically for FPGA-homed cache lines.* The aim of such an interface is to relive the applications from the burden of maintaining the state of FPGA-homed cache lines. Applications, without knowing the state of a cache line, can initiate local transactions on a cache line with the DC. Upon completion of the transaction, the DC provides certain guarantees on the state of the cache line while abstracting all the complexity involved in interacting directly with the coherence protocol.

The coherence transactions can be specified and modified depending on the needs of an application. The only consideration when defining coherence transactions is the following. The design choices we have made in the previous chapters (takeaway 5.2) prevent applications on the FPGA from caching FPGA-homed cache lines (HS of a cache line is always I). As a result, applications on the FPGA are required to interact with the DC through *coherent non-caching* transactions.

Takeaway 7.2. *Users are free to define coherence transactions based on the needs of an application. The only caveat being these operations should be coherent non-caching operations.*

In this chapter, we will look at defining and redefining DC's interface to suit different applications. In this process, we will define local coherence transactions (section 7.2) and see how an application that has no information on the state of a cache line can extract certain guarantees using these transactions. We also look at defining coherence messages for local transactions (section 7.3) and how to classify them into separate message classes to avoid deadlocks. Once the semantics of coherence transactions are decided, we will look at how to specify them for the state space exploration tool to automatically generate the state machine (section 7.4).

In detail, we first specify a set of coherence transactions that allow applications to clean or clean-invalidate FPGA homed cache lines from CPU's LLC (section 7.6). Then we modify this specification to include locking capabilities for cache lines (section 7.7) to demonstrate the flexibility of this approach and to discuss things to be considered when building your own set of transactions (for example, avoiding deadlocks and starvation). We then look at a sample application layer protocol that interacts with the DC to extend the notion of coherence across multiple cache lines (section 7.8). We also specify certain coherence transactions that can be used for resource maintenance (section 7.9) and are not visible to the application layer. Finally we look at at the *message packet format* for all local coherence messages as they are currently implemented (section 7.11).

It is to be noted that the DC interface shown in this chapter is not specific to Enzian but can be applied to any symmetric coherent system with aforementioned assumptions on the coherence protocol and interconnect.

7.2 Local Coherence Transactions

How applications can interact with the DC is not defined by CPU's coherence protocol. As a result, we have complete control on what coherence transactions would be exposed by the DC to applications. In this section, we focus on a DC interface that provides two specific transactions namely *local-clean* and *local-clean-invalidate* coherence transactions. The semantics of these transactions are as follows.

Local-clean transaction: An application can initiate a local-clean transaction on a cache line by issuing a local-clean request to the DC. This would cause the DC to clean (downgrade to S state) the cache line if it is dirty in CPU's LLC. Once the cache line has been cleaned, the DC responds to the application with an acknowledgment indicating that the transaction has been completed.

Local-clean-invalidate transaction: A local-clean-invalidate transaction for a cache line causes the DC to clean and invalidate the cache line (downgrade from E/M or S to I) if it is present in the CPU's LLC. This is also a request-response interaction.

These two transactions were chosen for implementation as a lot of other coherent non-caching operations can be built using these operations and application's access to DC's memory transactions as described later in section 7.10.

It is also to be noted that the applications do not have to worry about the state of a cache line or, how to interact with the CPU when initiating a local coherence transaction. Once the request is issued by the FPGA application, the DC transparently ensures the semantics of the transaction before responding to the request. In the next section, we see the coherence messages that are involved in these transactions.

7.3 Local Coherence Messages

The coherence messages involved in local-clean and local-clean-invalidate transactions are described in this section. Depending on needs of applications, other local messages and transactions can be defined as well.

Local-clean request-response pair: The local-clean coherence transaction can be initiated by issuing a local-clean request (LC) to the DC. Upon completion of the transaction, the DC responds to the application with a local-clean-acknowledge (LCA).

Local-clean-invalidate request-response pair: The local-clean-invalidate coherence transaction can be initiated by issuing a local-clean-invalidate request (LCI) to the DC. Upon completion of the transaction, the DC responds to the application with a local-clean-invalidate-acknowledge (LCIA).

7.4 Local Coherence Message Classes

Similar to the interconnect's message classes in subsection 4.4.1, local coherence messages are also classified into message classes with each message class having a dedicated VC to exchange coherence messages. These message classes are not part of the message classes offered by the interconnect but are created specifically to support interaction with application layer. The reason for classifying coherence messages in to different message classes and VCs is to avoid deadlocks as discussed in section 5.5.

Only the classes of messages involved in local-clean and local-clean-invalidate transactions are defined here. Additional message classes can be defined depending on the local interface exposed by the DC. The numbers assigned to the VC are arbitrary and should not overlap with VC numbers assigned to the interconnect's message classes (ECI VC numbers in our case). The local coherence message classes are described below and the local coherence messages along with their message classes are summarized in Table 7.1.

Local-forward-request-without-data message class: This message class consists of requests for local clean and clean-invalidate transactions, namely LC and LCI messages. VCs 16 and 17 are assigned for this message class. Following the same convention as ECI, odd VCs carry events for even cache line indices and vice versa. This odd/even split is done only for performance reasons.

Local-response-without-data message class: This message class consists of acknowledgments from the DC for previously issued requests by applications, namely LCA and LCIA messages. VCs 18 and 19 are assigned for this message class.

Message Class	VC #	Message
Local-forward-request-without-data	16, 17	LC, LCI
Local-response-without-data	18, 19	LCA, LCIA

Table 7.1: *Local message classes, their VC numbers and associated messages that can be used by FPGA applications to initiate clean and clean-invalidate transactions.*

7.5 Specification of Local Coherence Transactions

As seen in subsection 4.6.2, all transactions with the remote CPU node is specified in the form of state equations. Although such a representation could be extended to local coherence transactions, for the sake of simplicity, these transactions are specified through two look-up tables: one to specify when a local coherence transaction starts and another to specify when the local transaction completes. Specifying local transactions through look-up tables is quick but prone to errors. Coming up with a rigorous way of specifying local coherence transactions is part of future work.

Both local-clean and clean-invalidate transactions can potentially cause the DC to initiate a forward-downgrade transaction with the remote node through the coherent interconnect. We had seen in section 6.2 how the RS of the cache line in the DC's directory is used to keep track of coherence transactions with the remote node. But tracking just remote transactions are not enough, ongoing local transactions would have to be tracked in the DC's directory as well. Since the HS of the cache line is always I and unused, we modify the HS to indicate that a local transaction is ongoing.

Design Choice 7.1. *The HS of a cache line in the DC's directory is used to keep track of ongoing local transactions and RS is used to keep track of ongoing coherent transactions with the remote node.*

Table 7.2 shows the initial conditions for a local coherence request to be consumed by the DC. Till the initial conditions are met, the local request would be stalled by the DC. In this table, the RS of a cache line indicates when a local request would be consumed. When a request is consumed, the next HS and RS for the cache line is given in the table along with the action to be performed. This table does not specify when a local transaction is to be completed.

For example, in the first line, the local request LC is received when RS of the cache line in 1 (Invalid). Since the cache line is invalid in the CPU's LLC, the cache line is clean so the local request (LC) is consumed and a response LCA is sent to the application. There is no change in both HS and RS of the cache line after the DC performs the action. The same reasoning can be applied for the second and third lines of the table.

In the fourth line, a local clean-invalidate request is being handled when the RS of the cache line is S (2). Since the cache line has to be downgraded from S to I state in the CPU's cache, the DC issues an F21 (Action column in Table 7.2) coherence request to the CPU. The DC transitions the RS to 1_A21 (Next RS column in Table 7.2) to indicate that a forward downgrade transaction is in progress. It also transitions that HS (Next HS column in Table 7.2) of the cache line to intermediate state *1pCI* (final state 1 pending clean invalidate) indicating that a local clean-invalidate transaction is in progress. The progress of the forward-downgrade transaction with the CPU is tracked in the RS according to its specification described in chapter 6.

RS	Local Request	Next HS	Next RS	Action
1 (I)	LC	1	1	Send LCA
1 (I)	LCI	1	1	Send LCIA
2 (S)	LC	1	2	Send LCA
2 (S)	LCI	1pCI	1_A21	Send F21
3 (E/M)	LC	1pC	2_A32d	Send F32
3 (E/M)	LCI	1pCI	1_A31d	Send F31

Table 7.2: *Conditions to initiate local coherence transactions: Given the RS that identifies when a local request is to be handled, what should be the next HS, RS and action to be performed by the DC.*

From lines 1 to 2 in Table 7.2, it is to be noted that both local clean and clean-invalidate transactions would not cause an already invalid cache line (invalid in CPU's cache) to transition to an intermediate state. Transitions only happen to cache lines that are already cached in the CPU's LLC. This is important because both clean and clean-invalidate requests do not require an empty spot in DC's directory when the cache line is not cached in the CPU.

Table 7.3 gives the conditions that specify when a local transaction is completed. The status of when a cache line is clean or clean-invalid is defined by its RS. Hence, RS of the cache line identifies when an ongoing local clean or clean-invalidate transaction completes. For example, in the first two lines of Table 7.3, we have a specification for an on-going local-clean transaction as indicated by the intermediate HS, 1pC. When the RS of the cache line is *effectively* S or I, the local transaction is completed by issuing the LCA response and transitioning the HS of the cache line from 1pC to 1 (I, Next HS).

Thus the semantics provided by local coherence transactions have been specified. This specification is used by the state space exploration tool to schedule and complete local coherence transactions. This specification can also be expanded to other coherent non-caching transactions.

HS	Effective RS	Next HS	Action
1pC	2 (S)	1 (I)	Send LCA
1pC	1 (I)	1 (I)	Send LCA
1pCI	1 (I)	1 (I)	Send LCIA

Table 7.3: *Conditions to complete local coherence transaction: Given the HS and effective RS that identify when a local coherence transaction is completed, what should be the next HS and action to be performed by the DC.*

7.5.1 Effective remote state

The RS of a cache line can be one of the stable states: I, S, E/M or one of the many intermediate states. Each intermediate state can be assigned an *effective* stable state based on coherence messages that are in transit. The effective state of an intermediate state corresponds to the highest stable state that can be deduced with certainty given the coherence messages that are received and those in transit when the intermediate state was created.

For example, consider intermediate RS, 1_V32 that was created in state equations 5.39. From the state equation, we can deduce that a voluntary downgrade from E to S is in transit when the intermediate state is reached, which makes the cache line to effectively still be in E state in the CPU.

Similarly, the intermediate RS 2_WDDA from equation 5.33 has a write transaction in progress and the effective state of 2_WDDA is still E/M till the write transaction is completed.

The idea of effective state is a heuristic and is used instead of maintaining epoch for cache lines. In hindsight, maintaining epochs for cache lines would be the correct way of identifying when local transactions complete.

7.6 Guarantees Provided at DC's Application Interface

As stated previously, one of the goals of DC is to allow an application to infer certain guarantees on the state of cache lines in CPU's LLC by interacting with the DC. In this section we look at what guarantees the application can infer by observing the local and memory interfaces of the DC. The applications on the FPGA can interact with the DC state machine through the local clean and clean-invalidate transactions. In addition, applications can also observe and intercept memory transactions from DC.

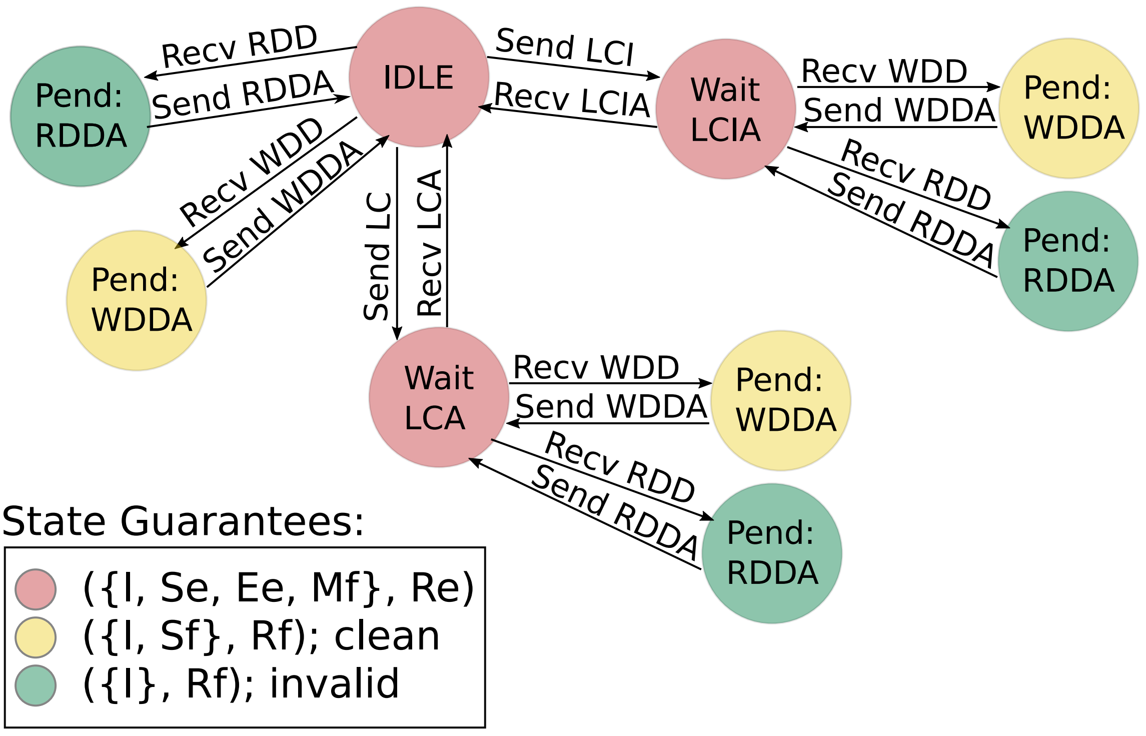


Figure 7.1: *State guarantees provided to applications by the DC through its local and memory interfaces at the granularity of a cache line.*

Figure 7.1 shows what an application can infer as RS of a cache line and where the most up-to-date data for the cache line resides by observing the coherence events issued by the DC protocol. Each circle represents an application state with certain guarantees represented in the format of $(\{possible\ RS\}, epoch\ of\ data\ in\ FPGA\ memory)$.

A "red" state implies the application has no information on the state of the cache line as indicated by the state-guarantee $(\{I, Se, Ee, Mf\}, Re)$ in the legend. That is, the RS of the cache line can be *invalid* (I), or *shared with FPGA memory having the most up-to-date copy* (Se), or *exclusive with FPGA memory having the most up-to-date copy* (Ee), or *modified with CPU LLC memory having the most up-to-date copy* (Mf). The suffixes "*e*" and "*f*" indicate different versions of the cache line data with suffixt f being the most up-to-date copy. For example, if the RS of the cache line is *modified*, the most up-to-date copy is present in CPU's LLC (indicated by Mf) and the FPGA memory has a stale copy (indicated by Re).

An "yellow" state implies the application can infer the RS to be either I or S with both CPU's LLC and the FPGA memory having the most up-to-date copies. This is indicated by the state-guarantee $(\{I, Sf\}, Rf)$.

A "green" state implies that the application can infer the RS of the cache line to be I with the FPGA memory having the most up-to-date copy as indicated by state-guarantee $(\{I\}, Rf)$.

To begin with, the application is in *IDLE* state where it does not have any information on the state of a cache line. At the *IDLE* state, the application might intercept a read request (RDD) that is issued by the DC. From the DC protocol model (section 4.2) we know that a read request will be issued by the DC only when the CPU makes an upgrade request for a cache line that is not cached in its LLC. Thus by observing the read request on a cache line, the application can infer that the state of the cache line in the CPU's cache is I and that the FPGA memory has the most up-to-date copy as indicated by legend ({I}, Rf). This continues to be the case till a response (RDDA) is sent to the read request, at which point there can be no more guarantees on the state of the cache line. This is shown by the transition between *IDLE* and *Pend:RDDA* states in Figure 7.1.

From the *IDLE* state, the application might also intercept a write request (WDD) for a cache line from the DC. Write requests are issued by the DC only when dirty cache lines are written back from the CPU's cache. Furthermore design-choice 5.2 guarantees that upgrades for this cache line would be stalled till the memory write transaction for this cache line completes. As a result, intercepting a write request, the application can infer that the state of the cache line is either S or I in the CPU's LLC (never E/M) and the data being written is the most up-to-date value for this cache line (indicated by *Rf*). This is shown in the transition between *IDLE* and *Pend:WDDA* states in Figure 7.1.

Instead of passively intercepting memory transactions, the application can also initiate a local-clean or local-clean-invalidate transactions from the *IDLE* state. From Table 7.3 it can be seen that the response to both clean and clean-invalidate requests are sent only when the state of the cache line in the CPU, matches the semantics of the transaction. That is, response LCA will be sent only if the state of the cache line is either S or I, and LCIA is sent only when the state of the cache line is I. However, this does not guarantee that the cache line continues to remain in this state when the response is eventually received by the application. For example, the CPU could have upgraded the cache line while the clean or clean-invalidate response is in transit.

This is shown by transitions from *IDLE* to *Wait LCA* and *Wait LCIA* states in Figure 7.1. The application can issue a clean or clean-invalidate request before waiting for a response. While waiting for the response, applications can observe memory transactions in order to infer guarantees discussed above. But there are no guarantees on the cache line's state or the location of the most up-to-date copy when the response is actually received.

These conditions imposed by the application interface of the DC might not be suitable for all types of applications. For example, certain applications would want the semantics of clean or clean-invalidate transactions to hold even after the responses are received. This further highlights the importance of having a flexible interface to provide different types of guarantees to the application. Next section discusses how the application interface can be modified to provide different guarantees.

Takeaway 7.3. *Having a flexible local interface for applications allows for the DC to provide different guarantees for different types of applications.*

7.7 Modifying DC's Application Interface For Locking Capabilities

In this section, we want to modify the guarantees that the DC provides to applications. Specifically, the clean and clean-invalidate transactions would be modified in such a way that the semantics of these transactions hold even after transaction responses are received and till the application relinquishes control. The aim of this section is to show that the application interface to the DC protocol is highly customizable and can be modified easily by changing the specification of local coherence transactions.

For example, in a clean-invalidate transaction the application issues a clean-invalidate request for a cache line. The DC should ensure that the CPU does not have a copy of the cache line and the most up-to-date copy resides in FPGA memory before sending a response to the application. When the clean-invalidate transaction completes, the DC *locks* the cache line from further upgrades by the CPU, before sending a response to the application. When the application receives the response, the cache line is guaranteed to be in I state in CPU's cache till the application specifically unlocks the cache line using an unlock coherence message.

Similarly, for clean transaction, the application issues a clean request to the DC. The DC ensures that the CPU has a read-only copy and the FPGA memory has the most up-to-date copy before responding to the application. The DC then completes the clean transaction by locking the cache line and responding to the application. When the application receives the response, it can infer that the cache line is either in S or I state in the CPU's LLC and the FPGA memory has the latest copy. The cache line continues to remain in this state till application issues a message to unlock this cache line to the DC.

Unlock transaction: The clean and clean-invalidate transactions, in addition to the semantics described in section 7.2, lock the cache line from being upgraded by the CPU. We introduce an *unlock* transaction where a single unlock message can be issued by the application to unlock a previously locked cache line. The unlock transaction is a posted transaction with a single response event and no request event.

7.7.1 Local coherence messages for locking capabilities

In addition to the local coherence messages defined in section 7.3 for clean and clean-invalidate transactions, we define an additional local coherence message for the unlock transaction.

Unlock response (UL): The unlock coherence message is issued by the application to the DC in order to unlock a cache line that was locked by a local-clean or local-clean-invalidate transaction. This coherence message is treated as a response (i.e. cannot be stalled) and have no requests associated with it.

7.7.2 Local coherence message classes for locking capabilities

We continue to have the message classes described in section 7.4 for clean and clean-invalidate transactions, namely *local-forward-request-without-data* and *local-response-without-data*. The unlock message is classified as a response and does not have any cache line data associated with it. So the Unlock message also belongs to *local-response-without-data* message class. Table 7.4 modifies the table in Table 7.1 to include the unlock message.

Message Class	VC #	Message
Local-forward-request-without-data	16, 17	LC, LCI
Local-response-without-data	18, 19	LCA, LCIA, **UL**

Table 7.4: *Local message classes and associated messages for FPGA applications to initiate clean-lock, clean-invalidate-lock and unlock transactions.*

7.7.3 Specification of local coherence transactions for locking capabilities

Section 7.5 describes how local coherence transactions are specified using two look-up tables. These tables are modified here to incorporate locking capabilities.

Table 7.5 shows the initial conditions that indicate when local coherence transactions can be initiated by the DC (modifications made to Table 7.2 are highlighted in bold). From the first three lines it can be seen that the HS of the cache line transitions to intermediate state *1pUL (1 pending unlock)*, indicating a lock, as soon as the response to a clean or clean-invalidate request is sent. The state space exploration tool stalls any upgrade requests from the CPU when the cache line is locked. For remaining scenarios, clean and clean-invalidate transactions continue as previously described in section 7.5.

RS	Local Request	Next HS	Next RS	Action
1 (I)	LC	**1pUL**	1	Send LCA
1 (I)	LCI	**1pUL**	1	Send LCIA
2 (S)	LC	**1pUL**	2	Send LCA
2 (S)	LCI	1pCI	1_A21	Send F21
3 (E/M)	LC	1pC	2_A32d	Send F32
3 (E/M)	LCI	1pCI	1_A31d	Send F31

Table 7.5: *Conditions to initiate local coherence transactions: for clean-lock, clean-invalidate-lock and unlock transactions.*

Table 7.6 gives conditions under which a local transaction is deemed to be completed. From the table it can be seen that as soon as the clean or clean-invalidate transaction completes, the cache line gets locked by transitioning the HS of the cache line to *1pUL*.

Finally, whenever home state is 1pUL and unlock message (UL) arrives, the HS transitions to 1 (I) in the DC's directory. It is to be noted, since we do not explicitly specify what

HS	Effective RS	Next HS	Action
1pC	2 (S)	**1pUL** (I)	Send LCA
1pC	1 (I)	**1pUL** (I)	Send LCA
1pCI	1 (I)	**1pUL** (I)	Send LCIA

Table 7.6: *Conditions to complete local coherence transaction: for clean-lock, clean-invalidate-lock and unlock transactions.*

would happen when an unlock message is received for a non-locked cache line, the state space exploration tool assumes that this scenario is not allowed when generating the state machine.

7.7.4 Guarantees provided at DC's modified application interface

Figure 7.2 shows the guarantees provided by the modified application-interface with locking capabilities. It follows the same nomenclature provided in section 7.6. Applications that do not have any information on the state of a cache line can make a clean or clean-invalidate request to the DC. Once the DC has performed necessary actions, it locks the cache line and responds to the application. The state of the cache line is guaranteed to be unaltered when the application receives the response.

The application can proceed to make any modifications to the cache line directly in the FPGA memory before unlocking the cache line. All the modifications made during this period are atomic from the perspective of the CPU. Once the cache line is unlocked, the application does not have any guarantees on the state of the cache line.

7.7.5 Starvation in the modified DC interface

Though the locking capabilities of the modified DC interface are useful, it suffers from the problem of starvation. It was noted in section 7.5 that for the local coherence transactions *without* any locking capabilities, only the states of the cache lines that are cached in the CPU would require a spot in the directory. As such, the local clean and clean-invalidate transactions would never occupy an empty spot in the directory. Applications on the FPGA can only cause cache lines to be downgraded which can potentially free up directory resources.

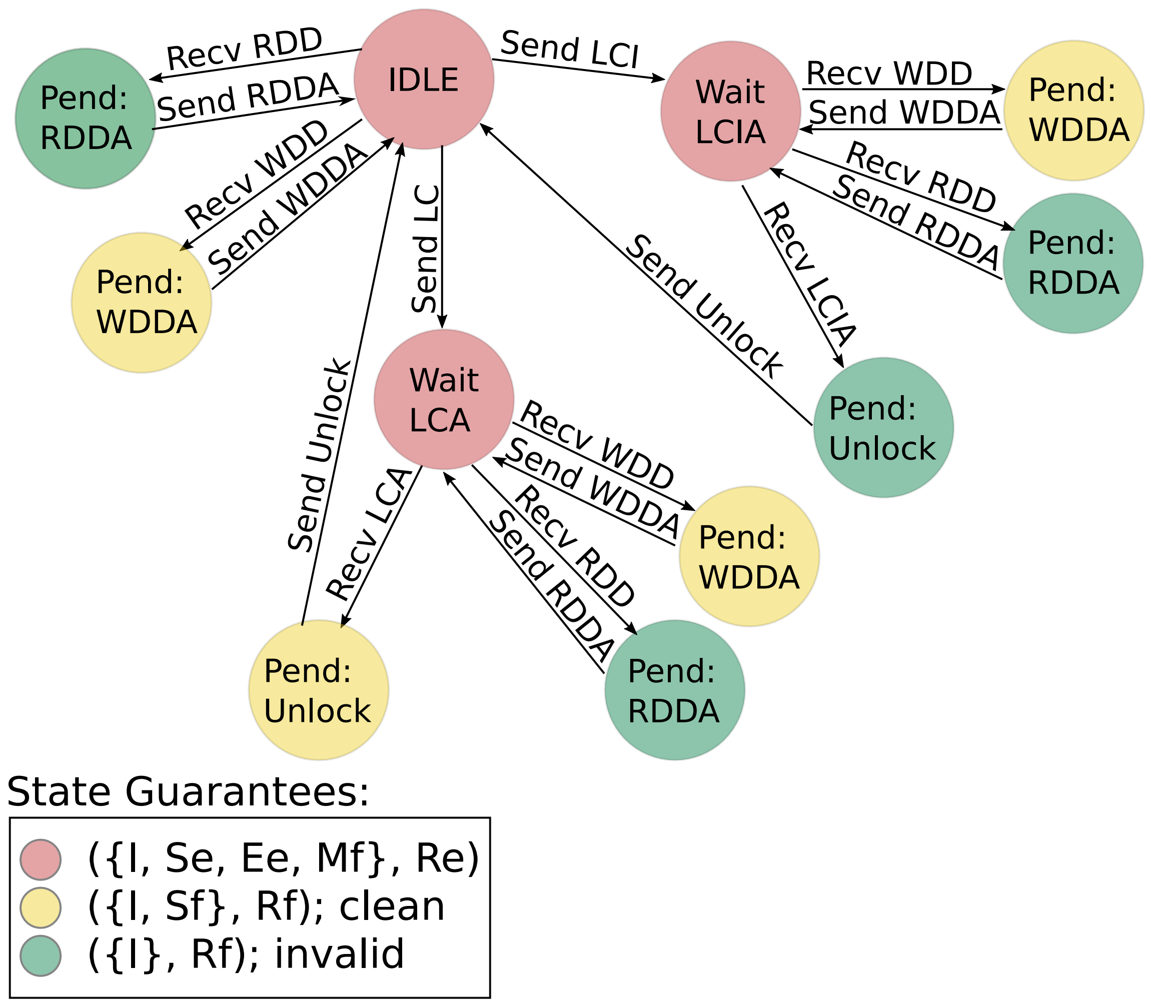


Figure 7.2: *State guarantees on a cache line that can be inferred by FPGA applications for modified DC application interface with locking capabilities.*

But for the interface with locking capabilities, it can be seen from lines 1 and 2 of Table 7.5, locking a cache line that is not cached in the CPU would require an empty spot in the directory to store the intermediate (locked) state. This allows the application to fill up the directory with locked cache lines while upgrade requests by the CPU, even for non-locked cache lines, starve waiting for a spot.

To overcome this issue, there should be additional types of transactions from the DC that should cause the application to unlock cache lines, instead of passively waiting for the application to voluntarily unlock.

This problem was discovered when running experiments where the CPU and multiple threads on the FPGA where concurrently accessing data. Although this problem would exist irrespective of the number of FPGA threads, it was observed that this problem is more pronounced when running a large number of FPGA threads in order to maximize memory bandwidth utilization. Fixing this problem would be part of future work.

Takeaway 7.4. *The DC protocol layer's application interface, the coherence transactions it allows and the guarantees it provides can be changed easily by changing the specification of local coherence transactions. These changes can be taken by the state space exploration tool to generate new protocol variants automatically. This can be useful for application developers to tailor the DC's protocol to simplify the application layer protocol. That said, care must be taken when defining your own local coherence transactions to avoid deadlocks, livelocks and starvation.*

7.8 Application Layer on top of DC Protocol Layer

In this section we will see how to define the application layer protocol in its interaction with the DC and memory transactions and considerations to make it deadlock free. Note just like how DC protocol is different from its implementation, the application layer protocol is also different from its implementation. So far we have seen that the DC protocol layer provides a simplified interface for the application layer to interact with the coherence protocol: An application can intercept memory transactions and initiate local transactions. The DC protocol layer also guarantees that coherence transactions between different cache lines would be independent of each other. This implies that the application layer can, for example, stall a memory transaction one cache line (*cache line A*) and initiate a local transaction for a different cache line (*cache line B*) and expect the local transaction on *cache line B* to complete before allowing the stalled memory transaction on *cache line A* to continue.

This allows applications on the application layer to make associations and guarantee invariants across different cache lines which can be useful in extending the idea of coherence to software on the CPU. To illustrate this let us consider how an application guarantee the following invariant: Whenever *cache line A* is cached in CPU's LLC we want *cache line B* to not be cached in CPU's LLC and vice versa. That is if RS of *cache line A* is S or E/M, RS of *cache line B* should be I and vice versa.

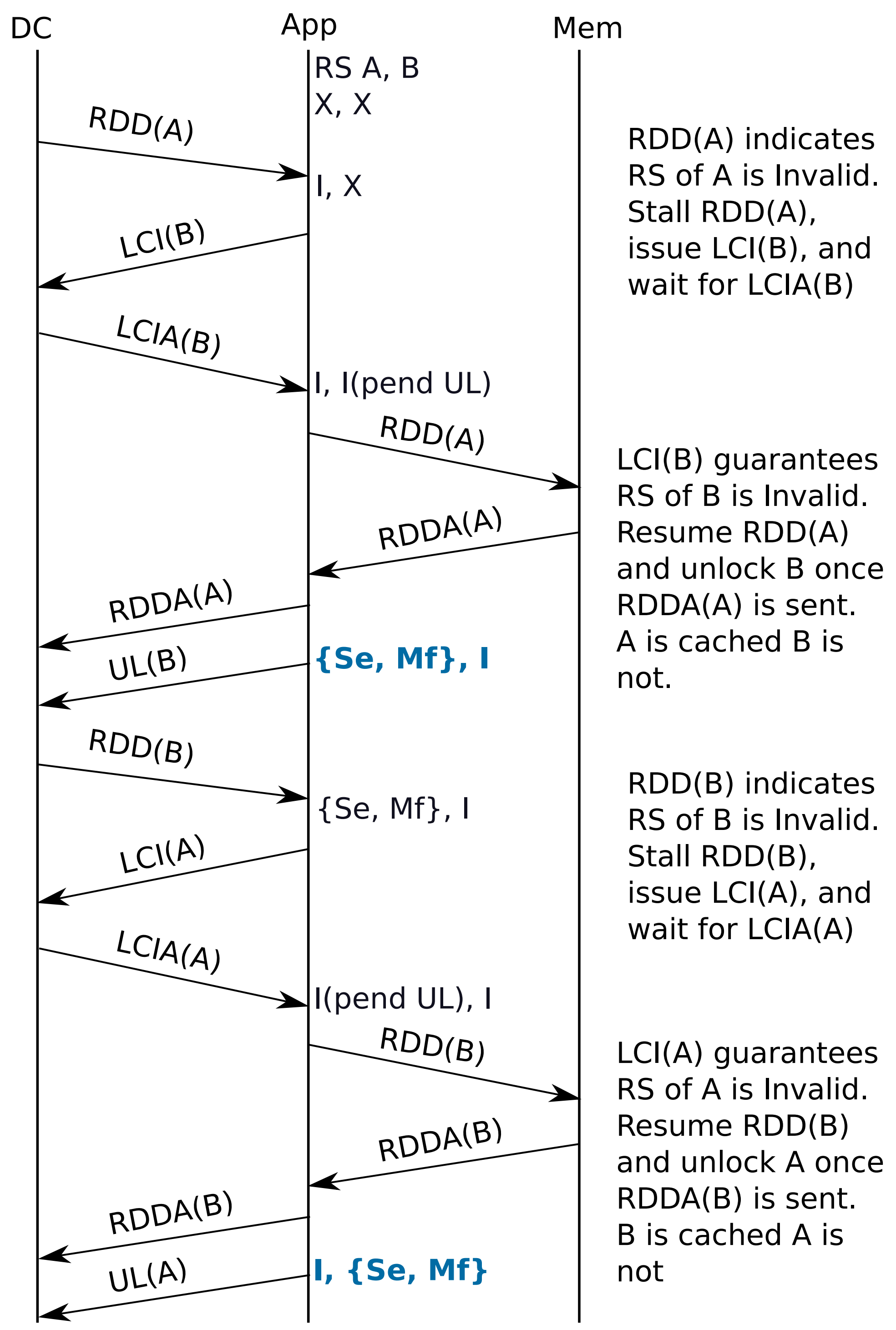

DC
App
Mem
RS A, B
X, X
RDD(A)
I, X
RDD(A) indicates RS of A is Invalid. Stall RDD(A), issue LCI(B), and wait for LCIA(B)
LCI(B)
LCIA(B)
I, I(pend UL)
RDD(A)
LCI(B) guarantees RS of B is Invalid. Resume RDD(A) and unlock B once RDDA(A) is sent. A is cached B is not.
RDDA(A)
RDDA(A)
{Se, Mf}, I
UL(B)
RDD(B)
{Se, Mf}, I
RDD(B) indicates RS of B is Invalid. Stall RDD(B), issue LCI(A), and wait for LCIA(A)
LCI(A)
LCIA(A)
I(pend UL), I
RDD(B)
LCI(A) guarantees RS of A is Invalid. Resume RDD(B) and unlock A once RDDA(B) is sent. B is cached A is not
RDDA(B)
RDDA(B)
I, {Se, Mf}
UL(A)

Figure 7.3: *Application layer protocol that guarantees when cache line A is cached in the CPU's LLC, cache line B is not cached and vice versa. The application can infer remote states of cache lines A and B by exchanging coherence messages with the DC.*

Let us assume initially *cache line A* is not cached in the CPU's LLC. When the CPU wants to cache *cache line A*, the DC would generate a memory read request (RDD) for *cache line A*. The DC also guarantees that the RS of *cache line A* would be I till a memory read response is received (see *Pend:RDDA* state in Figure 7.2).

The application can intercept the memory read request on *cache line A* and stall it. Meanwhile, the application can initiate a clean-invalidate-lock local transaction (LCI) on *cache line B*. Since *cache lines A and B* are different cache lines, we can expect the DC to complete the LCI transaction on *cache line B* even when read response (RDDA) on *cache line A* is pending. Once response LCIA is received for *cache line B*, the DC guarantees that RS *cache line B* is locked at I (See *Wait LCIA* transition in Figure 7.2) at which point, the application can respond to the read request for *cache line A* and unlock *cache line B*. The RS of *cache line A* would be S or E/M, and the RS of *cache line B* would be I till a memory read request is observed for *cache line B*. This application layer protocol is shown in for both cache lines A and B in Figure 7.3.

It is also to be noted that the DC does not guarantee independence of coherence transactions on a single cache line even if the coherence transactions are different. For example, a memory transaction and local transaction on a cache line are not guaranteed to be independent. This means that if a memory request (RDD or WDD) on *cache line A* is stalled before issuing a local transaction on the same cache line, it is not guaranteed that the local transaction on *cache line A* would complete before the memory response (RDDA or WDDA) is sent. Assuming that the opposite can lead to application protocol deadlock. The conditions for application protocol deadlock are shown in Figure 7.4. For the implementation of the application protocol to be deadlock free, the application protocol itself has to be deadlock free, and in addition the implementation should also consider deadlocks that arise due to limited availability of resources.

Takeaway 7.5. *The DC guarantees independence of coherence transactions on different cache lines (each cache line is independent). This means an application can interrupt coherence transaction on one cache line to initiate and complete coherence transactions on other cache lines. But the DC does not guarantee independence of different transactions within a cache line, therefore interrupting a transaction to initiate a different transaction on the same cache line by an application when interacting with the DC can cause deadlocks.*

Now we have seen how applications can handle local coherence transactions. In the next section, we will see some coherence transactions that are used by the DC to perform

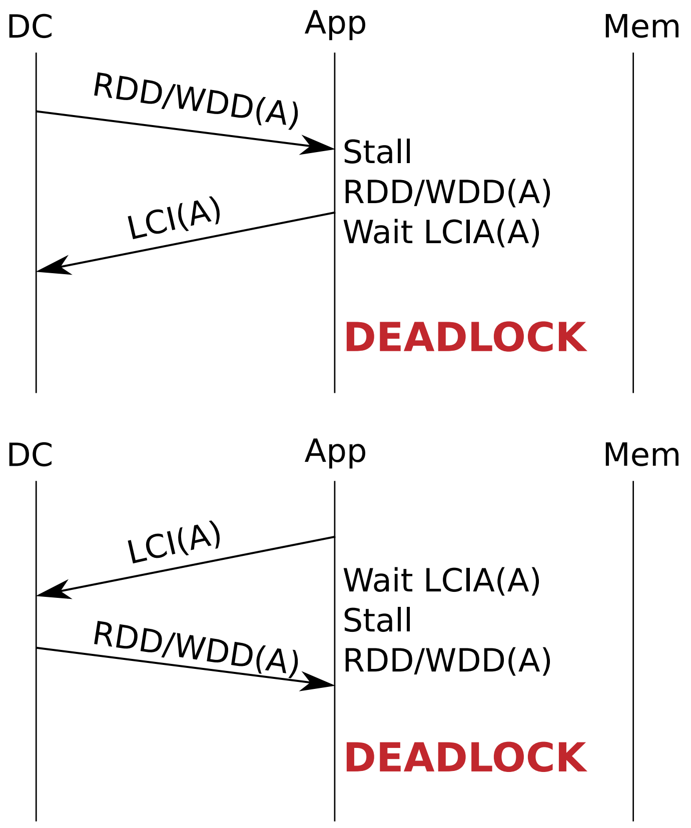


Figure 7.4: *Application protocol deadlock scenarios: Applications stalling coherence events on a cache line that are issued by the DC while initiating new coherence transactions on the same cache line can lead to deadlocks.*

resource maintenance, and are not exposed to applications.

7.9 Directory Maintenance Operations

The directory on the FPGA's DC is a limited resource. As such, there might be a need for the DC to maintain and free-up this resource whenever necessary. The directory holds the states of cache lines that are cached in the CPU. The DC would have to clean-invalidate previously cached cache lines from CPU's LLC if it has to free up directory resources.

The main caveat here that the clean-invalidate transaction is not initiated by any application on the FPGA but rather by the DC whenever it needs to perform resource maintenance. This means that directory maintenance operation is not initiated by a coherence event received through a VC but rather by some conditions within the DC. Also, the DC does not have to issue any coherence messages to indicate that the directory maintenance operation has completed.

For this reason, we introduce a special coherence transaction called "Induced-Clean-Invalidate (ICI)" transaction. It begins with the DC issuing an ICI request to the protocol state machine and the state machine along with a cache line address. The state machine clean-invalidates the cache line from the CPU's LLC. The transaction completes when the RS of the cache line is I and no coherence event is generated to indicate completion. Upon completion, the directory has an empty spot. This transaction is not visible to applications that interact with the DC and do not provide any guarantees to applications.

7.9.1 Specification of ICI transaction

The two table scheme described in section 7.5 is used to specify the induced-clean-invalidate transaction. Table 7.7 specifies what the state machine would have to do when an ICI event is invoked by the DC. If the RS of the cache line is already invalid, no specific action needs to be taken. If the RS of the cache line is S or E/M, a forward downgrade transaction is initiated and the RS of the cache line transitions to an intermediate state to keep track of the forward downgrade transaction. The HS of the cache line also transitions to intermediate state *1pICI* to indicate that an induced-clean-invalidate transaction is in progress.

RS	Local Request	Next HS	Next RS	Action
1 (I)	ICI	1	1	No Action
2 (S)	ICI	1pICI	1_A21	Send F21
3 (E/M)	ICI	1pICI	1_A31d	Send F31

Table 7.7: *Conditions to initiate induced-clean-invalidate (ICI) transaction.*

Table 7.8 shows when an induced-clean-invalidate transaction is completed. If the RS of the cache line is I, the transaction completes by transitioning the HS to I as well. No action needs to be performed by the state machine when the transaction completes.

HS	Effective RS	Next HS	Action
1pICI	1 (I)	1 (I)	No Action

Table 7.8: *Conditions to complete induced-clean-invalidate (ICI) transaction.*

7.10 Miscellaneous Local Coherence Transactions

In addition to local clean and clean-invalidate transactions, a few other local transactions are specified but have not been tested. Although these transactions are present in the protocol state machine, they are not supported by the current DC implementation. The reason for this that an application can implement these transactions by interacting with both DC and memory directly.

The first coherence transaction is “Local Read (LR)” which allows an application to perform a coherent non-caching read. Any dirty data is cleaned from the CPU’s LLC into the FPGA memory and provided as a response to the application. The state machine signals the end of the transaction using “Local Read Acknowledge (LRA)” coherence message.

With the current DC implementation, an application can do the same by clean-locking the cache line and reading the memory once the acknowledgment for the clean request is received. Once the memory is read, the application can unlock this cache line.

The second coherence transaction is “Local Write (LW)” which is allows the application on the FPGA to perform a coherent non-caching write. The protocol state machine, clean-invalidates any dirty data present in the CPU’s LLC and overwrites this with the data provided by the application. The state machine also signals the end of transaction to the application through “Local Write Acknowledge (LWA)” coherence event.

With the current DC implementation, an application can do the same by clean-invalidate-locking the cache line and writing to the memory once the acknowledgment for the clean-invalidation is received. Once the data is written, the application can unlock this cache line. These transactions are evaluated with the current DC implementation in subsection 9.6.2.

7.11 Local Events Packet Formats

In this section, the format of coherence messages exchanged between the application and DC are provided. The description of each field and behavior of DC are also provided.

7.11.1 LC and LCI request packet formats

local-clean (LC) and clean-invalidate (LCI) requests do not have any data associated with them. Similar to coherence headers in ECI, these messages are 64-bits wide and have the format shown in Table 7.9.

QWord	63:59	58:56	55:50	49:46	45	44	43:42	41:40	39:0
0	opcode [4:0]	xb3 [2:0]	hreq_id [5:0]	dmask [3:0]	ns [0:0]	xb1 [0:0]	rnode [1:0]	xb2 [1:0]	address [39:0]

Table 7.9: *Local clean and clean-invalidate packet format.*

The description of the fields are as follows:

- *opcode*: This 5-bit field along with the VC number is used to identify the coherence event and decode its bit-fields. As seen in Table 7.1, both local clean and clean-invalidates are assigned to VCs 16 and 17. The opcode for local-clean request is 0 (5'b00000) and opcode for local-clean-invalidate request is 1 (5'b00001).
- *xb3*: This field is 3 bits of don't-cares (Xs) and are ignored. Default value can be 0.
- *hreq_id*: This 6-bit field can be used by application to track the coherence transactions if necessary. The response associated with this request will be tagged with the same transaction ID. There are no ordering guarantees for multiple coherence transactions tagged with the same transaction ID. In other words, the order of responses received for these transactions will not match the order in which the requests were sent. The default value can be 0.
- *dmask*: This 4-bit field is currently unused but can be used to clean or clean-invalidate cache lines at sub-cache-line granularity. The default value can be 15 (4'b1111) indicating all sub-cache-lines.
- *ns*: This 1-bit field is used to indicate if the memory is non-secure. The default value can be 1 (1'b1).
- *xb1*: 1-bit of don't-care. The default value can be 0.
- *rnode*: This 2-bit field identifies the node that issues this request. The default value can be the node ID of the FPGA, which is 1 (2'b01).
- *xb2*: 2-bits of don't-care, default value can be 0.
- *address*: This is the 40-bit cache line physical byte address. In this case it would be an address in the FPGA address space.

7.11.2 LCA and LCIA responses packet formats

local-clean-acknowledge (LCA) and clean-invalidate-acknowledge (LCIA) responses do not have any data associated with them. Similar to coherence headers in ECI, these messages are 64-bits wide and have the format shown in Table 7.10.

QWord	63:59	58:56	55:50	49:46	45	44:40	39:0
0	opcode [4:0]	xb3 [2:0]	hreq_id [5:0]	dmask [3:0]	ns [0:0]	xb5 [4:0]	address [39:0]

Table 7.10: *Local-clean-acknowledge and clean-invalidate-acknowledge packet format.*

The description of the fields are as follows:

- *opcode*: The VC number and 5-bit opcode is used to uniquely identify a coherence message and decode its bit-fields. As seen in Table 7.1, both acknowledgments are assigned are assigned to VCs 18 and 19. The opcode for LCA is assigned opcode 0 (5'b00000) and LCIA is assigned opcode 1 (5'b00001).
- *xb3*: 3-bits of don't-cares, can be any value.
- *hreq_id*: 6-bits of transaction identifier. This will have the same value as the hreq_id of the request that initiated the transaction.
- *dmask*: Currently unused, will always return 15 (4'b1111) indicating that an entire cache line was cleaned or clean-invalidated.
- *ns*: Non-secure bit as returned by the CPU.
- *xb5*: 5-bits of don't-cares, can be any value.
- *address*: 40-bits of cache line physical byte address that was cleaned or clean-invalidated.

7.11.3 Unlock (UL) response packet format

The unlock response packet also has only a 64-bit header and does not have any data associated with it. The packet format is shown in Table 7.11.

The descriptions of the fields are as follows:

QWord	63:59	58:40	39:0
0	opcode [4:0]	xb19 [18:0]	address [39:0]

Table 7.11: *Unlock response packet format.*

- *opcode*: The VC number of 18 or 19 and an opcode of 2 (5'b00010) is used to uniquely identify an unlock response header and decode its bit-fields.
- *xb19*: 19-bits of don't-cares.
- *address*: 40-bit physical cache line byte address from the FPGA address space that has to be unlocked.

The format of the packets do not change between the locking and non-locking variants, just the guarantees provided changes.

7.12 Summary

In this chapter, we have seen how the DC protocol layer provides a simplified interface for the application layer to interact with the coherence protocol. Applications do not have to keep track of the state of cache lines but rather interact with the DC to initiate local transactions and get guarantees on the cache line state. All underlying coherent transactions with the remote node are handled by the DC thereby simplifying the application state machine.

We have also seen how local coherence transactions can be specified without starvation and deadlocks, and how different specification for local transactions affect the interface and guarantees exposed to the application layer.

Furthermore, we saw how different types of applications can rely on the guarantees provided by local coherence transactions, without having a need to modify the underlying protocol layer, while extending the notion of the coherence protocol to software on the CPU. Examples of such applications will be presented in chapter 9. Finally, we looked at coherence transactions that are defined for directory maintenance and not exposed to the application layer.

It is to be noted that the DC protocol and its application interface is highly customizable and can be changed by modifying their specification (takeaways 5.3 and 7.4). Care must be taken when modifying the specification to ensure that coherence invariants are maintained and that there are no deadlocks. The state space exploration tool can take this specification and automatically generate a state machine making it easy for application developers to quickly develop and deploy new protocol variants and application interfaces.

In the next chapter, we will look at how the DC protocol layer is implemented on the FPGA.

8 Distributed Directory Controller

8.1 Introduction

In section 4.1, we discussed about the two components of directory protocol layer, namely the *protocol state machine* and *its implementation*. Building the protocol state machine was extensively discussed in chapters 5, 6 and 7. In this chapter, we discuss about how it can implemented on the FPGA. Our implementation is tailored to the Enzian platform.

Based on the guarantees expected of the directory layer, the expectations of a generic DC implementation is as follows. The DC on the FPGA implements the protocol state machine to provide coherent access to the FPGA attached byte-addressable memory. Although the FPGA has limited resources, the DC should faithfully reproduce all guarantees provided by the protocol state machine. The DC should ensure that each FPGA-homed cache line has its own state machine (i.e. cache lines are mutually independent). It should also ensure that the transitions within a cache line are faithful to the state machine to maintain coherence invariatns (i.e. there should be no hazards during state transitions). Even if the protocol state machine is deadlock free and performant, it is not guaranteed that DC, its implementation on the FPGA, is deadlock free or performant. Deadlocks can arise due to limited availability of resources (resource deadlocks described in subsection 3.4.1). Hence these also would have to be considered when designing the DC.

Moreover, we want the generic DC implementation to be highly customizable. Previously we have seen that the protocol state machine is generated from a specification and by changing the specification, we can have a different state machine (check takeaway 5.3). Furthermore, we have also seen in section 7.7 how applications on the FPGA that interact with the DC can benefit from having a flexible interface. Thus it can be advantageous to have a flexible DC design that can accommodate different protocol state machines as well as have an application interface that can be expanded with only minor modifications.

In this chapter, we will look at how a DC with such properties can be implemented on the FPGA. The structure of this chapter is as follows.

1. Section 8.2 describes the interface of DC.
2. Section 8.3 provides an overview of the distributed DC's architecture with Directory Controller Slices (DCSs) and multiple Directory Controller Units (DCUs), along with the reasons for choosing such an architecture.
3. Section 8.4 goes into the implementation details of the DCS.
4. Section 8.5 goes into implementation details of the DCU and how it provides the guarantees required for the protocol layer.

Note 8.1. *In this chapter, the term DC refers to our implementation of a DC on Enzian.*

Note 8.2. *There was a version of DC that was built before the current version which had different design choices. These design choices lead to performance bottlenecks and the insights gained by experimenting with it has lead to the current set of design choices. Section B highlights the difference in design choices and Section section B.1 provides a summary of the insights gained for readers who want more information.*

8.2 DC Interfaces

Coherence controllers on the CPU interact with the DC through the coherent interconnect and controllers on the FPGA have a local interface on the DC to interact with it. In addition to other coherence controllers, the DC also deals with memory events, that is, issuing read/write requests to memory and receiving responses. Our implementation of the DC has an interface to Enzian's coherent interconnect (ECI), a local interface to FPGA applications and an interface to the memory as shown in Figure 8.1

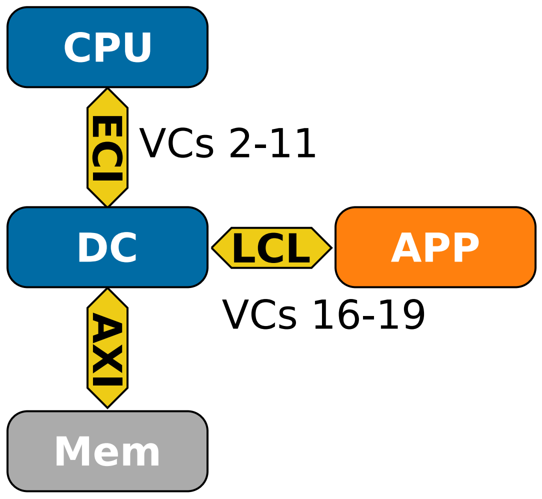


Figure 8.1: *DC interfaces: The DC has the ECI interface to interact with CPU, the local interface to interact with applications on the FPGA, and an AXI interface to interact with memory. Both ECI and local interfaces have a number of independent VCs for different message classes.*

Both ECI and local coherence messages are classified into message classes with each class having its own VC in order to avoid message-level deadlocks due to messages of different classes blocking each other (subsection 4.4.1). VCs are essentially FIFO buffers managed by credit-based flow control in ECI and by valid-ready flow control on the FPGA. Each VC has a number that can be used to identify the message class. Depending on the message class, a coherence message either contains a 64-bit coherence header and up-to 128-byte payload or only a 64-bit coherence header.

All coherence headers (ECI and local) have the following bit-fields in common: a 5-bit opcode, a 5/6-bit transaction ID, a 4-bit dmask (data mask) and a 40-bit cache line byte address. The opcode and the VC number are used to uniquely identify a coherence message and dmask is used to identify the size of the payload. In addition to these, there are also message specific bit-fields for different messages.

These VCs decouple individual coherence messages from the interface of the DC. That is, the DC does not have to provide a separate interface for each coherence message but have a standard VC interface for different *message classes*. The generic nature of such an interface allows the DC to provide a flexible local interface for applications on the FPGA.

Design Choice 8.1. *The applications on the FPGA interact with the DC through a standard VC FIFO interface. Similar to ECI, each message class has a separate VC* *interface. This implies that the local interface to the DC would not change if a new local coherence message is assigned to an existing message class. The interface would change only if new local message classes are added.*

Coherence controllers can initiate coherence transactions on a cache line, which are a chain of coherence messages towards a specific goal. For example, the DC can initiate a transaction with the CPU to clean-invalidate a cache line to which the CPU will respond. Each transaction is associated with a cache line address and is tagged with a transaction ID. All coherence messages in a transaction will have the same transaction ID and there can be multiple outstanding transactions on a cache line.

In addition to coherence transactions, there are also memory transactions on a cache line made up of AXI read request-response and write request-response pairs to a byte addressable memory. Memory transactions are also tagged with an ID which has no relation to a coherence transaction ID. More details on ECI, local and memory events are given in chapter 4 through chapter 7. In the next section, we will look at the directory of a DC and how it shapes the DC's architecture.

8.3 Overview of DC Architecture

8.3.1 DC directory sizing

In our system, there are no caching entities on the FPGA and CPU is the only entity that can cache FPGA-homed cache lines (see design choices in subsection 3.2.2). The state of these cache lines in the CPU's cache is tracked in the DC's directory. The LLC of CPU is 16-way set-associative and has a capacity of 16MB or 128K cache lines. By matching the size of DC's directory to the caching capacity of the CPU (the directory should be able to hold the states of at least 128K cache lines) conflict misses, arising due to the directory being full, and round-trip invalidations, to free up directory slots, can be avoided (see takeaway 2.1). More details on directories can be found in subsection 2.4.1.

Another implication of matching the directory size to the CPU's caching capacity is that the DC can piggy back on CPU's cache maintenance operations to maintain its directory resources, removing the need for the DC to have explicit directory maintenance operations. For example, when a set is full in the CPU's LLC, the set is also full in the DC's directory. In case the CPU wants to cache a new cache line in an already-full set, it has to create an empty spot in that set by evicting one of the previously cached cache lines. This eviction by the CPU will also empty a spot in the same set in DC's directory, there by freeing up directory resources.

The draw back of relying on the CPU to maintain directory resources on the FPGA is that it does not factor in any directory resource consumption by applications on the FPGA. For example, say a set in CPU's LLC is not full but the same set in the DC's directory is full due to applications on the FPGA occupying empty spots. When the CPU caches a new cache line, it would not perform any cache maintenance operation to free up resources in the full directory set, leading to a deadlock. This effectively limits the interaction between the DC and FPGA applications to transactions that do not require an empty spot in the DC's directory such as *clean* and *clean-invalidate* transactions (section 7.2). Any transactions with locking capabilities such a *clean-lock* or *clean-invalidate-lock* as seen in section 7.7 would require applications to use empty spots in DC's directory, which can potentially cause resource deadlocks. Note that the deadlock issue described here is different from the starvation issue discussed in subsection 7.7.5, the former is caused by not having directory maintenance operations on the DC whereas the latter is caused by incorrect protocol transaction design, although both are related to limited directory resources.

Design Choice 8.2. *For optimal performance, the size of DC's directory should be tuned to the charecteristics of the platfrom on which it is implemented. In our Enzian implementation, the directory can hold the state of up to 128K cache lines. This choice matches the size of the directory to the caching capacity of CPU's LLC (16 MiB) and avoids conflict misses, round-trip invalidations and explicitly maintenance of directory resources by the DC. Having the CPU maintain DC's directory resources simplifies DC design but limits what local transactions are allowed.*

Thus the directory of the DC can be viewed as a table where each set (row) has 16 ways (columns) and each cell stores the state of 1 cache line. There are 2^{13} sets to store states of all 128K cache lines. The cache line address defines the set into which the state of the cache line would be stored and the DC is free to place it in any of the *ways* available in this set. A tag is stored along with the state in a *way* to identify the cache line to which the state corresponds to. This is shown in Figure 8.2.

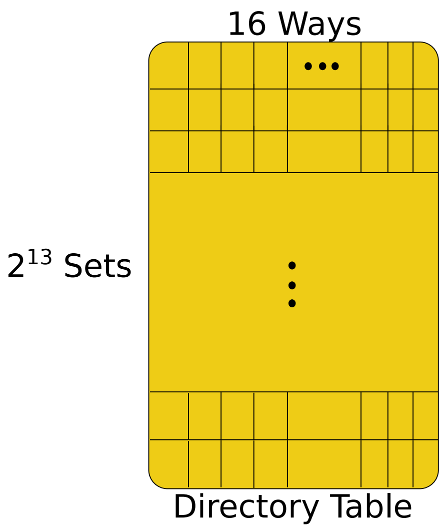


Figure 8.2: *DC directory can be viewed as a table where each cell stores the tag and state of an FPGA-homed cache line that is cached in the CPU's LLC. The size of the directory tracks the caching capacity of the CPU.*

8.3.2 Indexing into the DC directory

Although ECI supports addressing up to 1 TiB of memory, the DC exposes a 256 GiB of coherent address space on the FPGA. Thus ECI can have up to 40-bits of cache line byte-address, the DC uses only the lower 38-bits to address the byte addressable memory. The reasons for this would be explained in subsubsection 8.5.6.1.

Design Choice 8.3. *Although ECI allows addressing up to 1 TiB of FPGA memory, the DC exposes only 256 GiB of coherent address space on the FPGA. That is the AXI buses to the memory have 38-bit addresses. This design choice is done to optimize the resource consumption of Enzian's FPGA. Allowing access to full 1 TiB would lead to sub-optimal utilization of resources, increase utilization and add more congestion.*

The 38-bit cache line byte address is split into a *7-bit byte offset* (bits 6 to 0), followed by a *13-bit set index* (bits 19 to 7) and an *18-bit tag* (bits 37 to 18) as shown in Figure 8.3. The 7-bit byte-offset addresses one of the 128 bytes of data within a cache line. The 13-bit set index identifies which of the 2^{13} sets corresponds to this cache line. The remaining 18-bits are tag information that is stored in the directory along with the state in a *way* to identify the cache line corresponding to the cache line address. In the current configuration, the

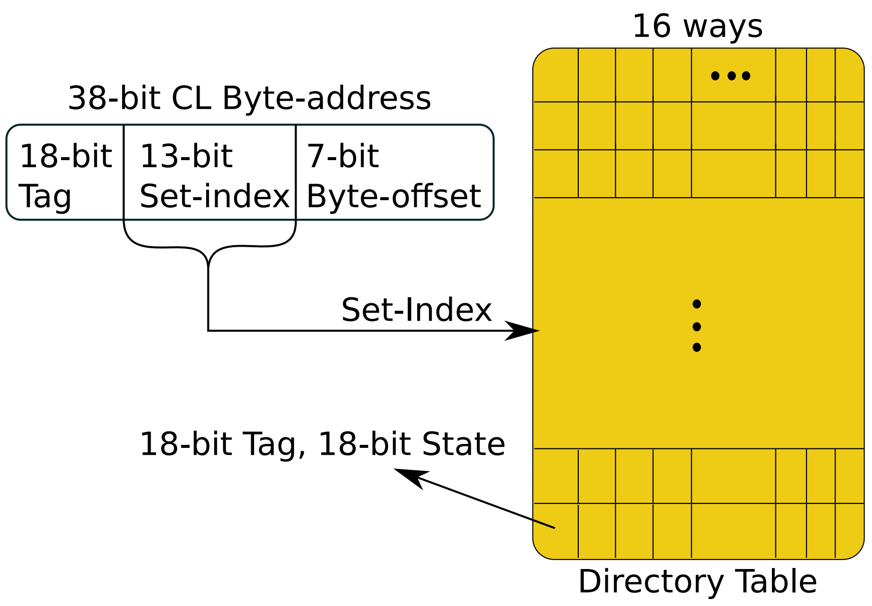


Figure 8.3: *Indexing into the directory using the cache line byte-address: The cache line byte-address contains a 13-bit set-index that is used to index into all ways of a particular set in the directory.*

state of a cache line is limited to 18 bits. Thus a specific way, storing 18-bit tag and an 18-bit state, is 36 bits wide as seen in Figure 8.3. The reason for this is that each row in an FPGA Block RAM (BRAM) is 36-bits wide and we would like to store contents of 1 *way* in a single BRAM row.

Design Choice 8.4. *The state of a cache line can be up to 18-bits in size (i.e. the protocol state machine can have a maximum of 262144 states). This choice is based on the fact that in the configuration we would like to use, each row in the BRAM is 36-bits wide (on Enzian's FPGA); enough to store an 18-bit tag and an 18-bit state. Our biggest DC protocol state machine has 77 states which is much less than the maximum limit.*

8.3.3 Directory controller units and slices

As seen earlier in takeaway 4.1, the state transitions on a given cache line is independent of the state transitions on other cache lines within the directory protocol. This enables us to split the directory across multiple DCUs with each DCU having a disjoint subset of the sets and combined together holds all sets in the directory.

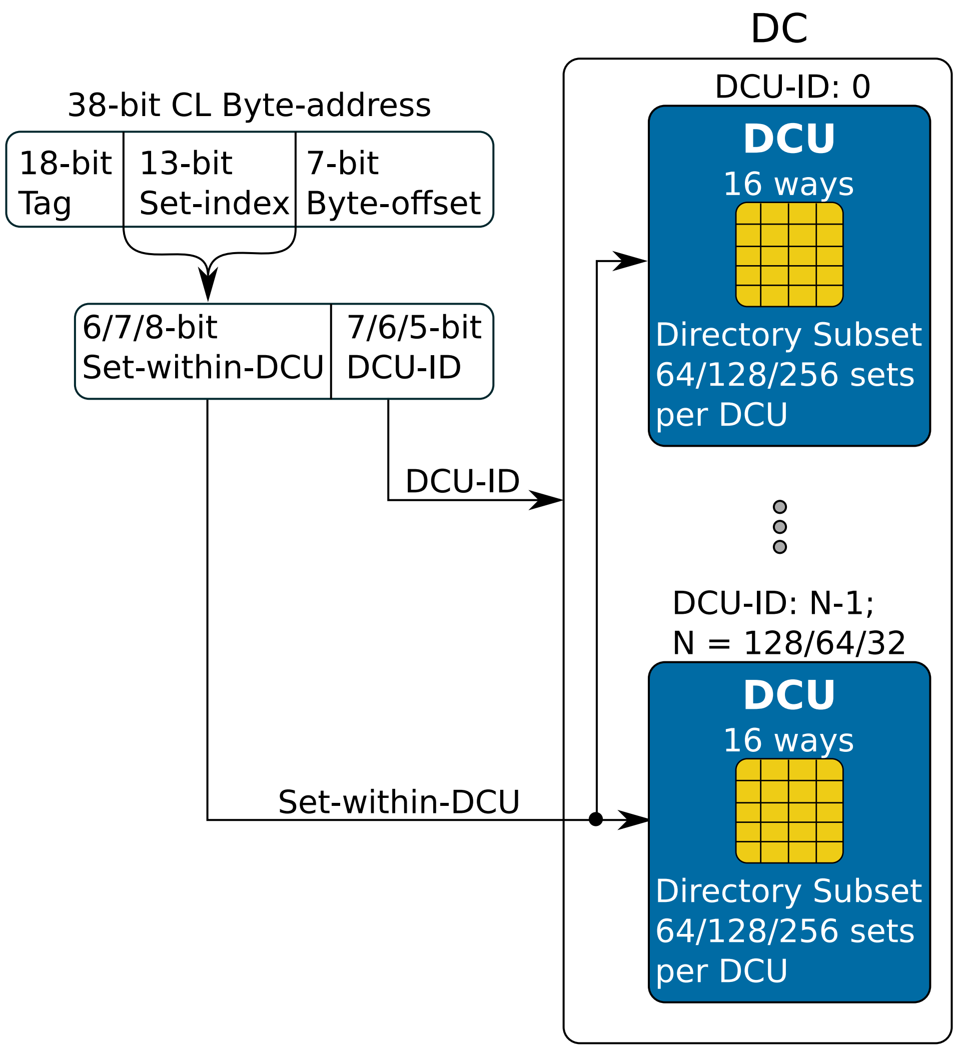


Figure 8.4: *DC is split into a number of parallel DCUs. Each DCU has a portion of DC's directory and can track a disjoint subset of cache lines.*

In current implementation, the number of sets within a DCU can be configured between 64/128/256 sets-per-DCU for reasons described in subsection 8.5.6. Thus to hold all 2^{13} sets we would need 128/64/32 DCUs depending on the number of sets-per-DCU. In order to index into the directory that is distributed across multiple DCUs, the 13-bit set index is further split into 7/6/5-bit *DCU Identifier (DCU-ID)* that chooses the DCU corresponding to this set and 6/7/8-bit set-within-DCU to identify the set within a given DCU as seen in Figure 8.4.

Furthermore, ECI allows for two independent channels of communication by providing separate sets of VCs for odd and even cache lines. To take advantage of this parallelism, the DCUs are organized evenly between odd and even DCSs as shown in Figure 8.5. Each DCS contains half the total number of DCUs (i.e. each DCS has 64/32/16 DCUs). The way this factors into directory indexing is that the DCU-ID is split into 1-bit odd/even DCS identifier and remaining 6/5/4 bits point to the DCU within a DCS (referred to as *DCU-Index (DCU-IDX)*). It is to be noted that even numbered VCs transport coherence messages for odd cache line indices. So the DCS connected to even VCs serve odd cache line indices and vice versa.

Design Choice 8.5. *The DC has 2 DCSs, one for odd cache line indices and the other for even. This is due to ECI having parallel channels of communication for odd and even cache lines. Each DCS contains 64/32/16 DCUs depending on the chosen configuration. Each DCU has a disjoint subset of the directory's sets with 64/128/256 sets per DCU and all 16 ways. This organization of the DC is based on the characteristics of ECI, Enzian and can vary between platforms.*

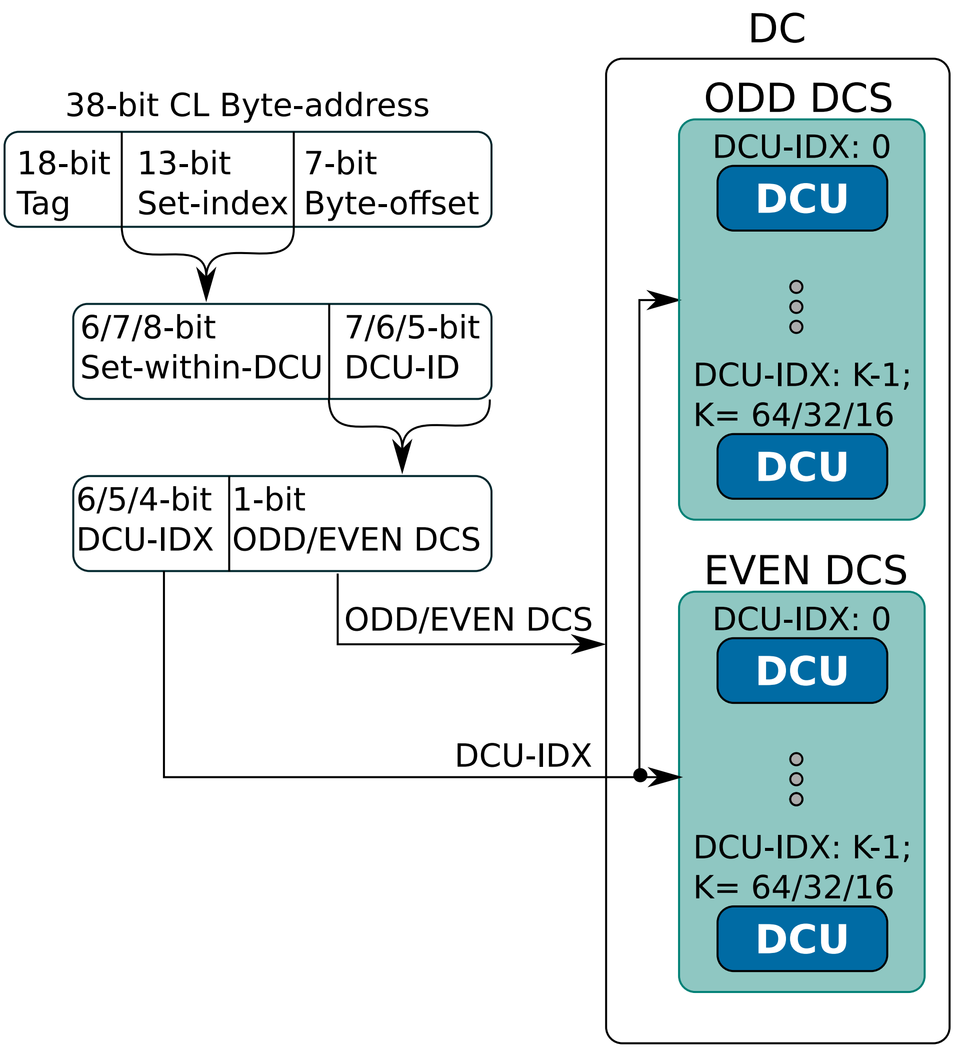


Figure 8.5: *DCUs are organized between odd and even DCSs. This is done to take advantage of parallelism offered by ECI which provides independent channels of communication for odd and even cache lines.*

8.3.4 Non-existent memory in address space exposed by DC

As mentioned earlier, the DC exposes a 256 GiB FPGA address space and has a directory big enough to match the caching capacity of the CPU. In order to optimize on the performance of the directory, a portion of the address space where the 18-bit tag is all 1s is considered to be non-existent. Why this "non-existent" region is needed within the FPGA address space is described in subsubsection 8.5.6.2. Thus we have a 1 MiB region in the end of the 256 GiB region that is considered to be non-existent as shown in Figure 8.6. Any access to this region would result in an error. The presence of such a region will depend on the platform characteristics.

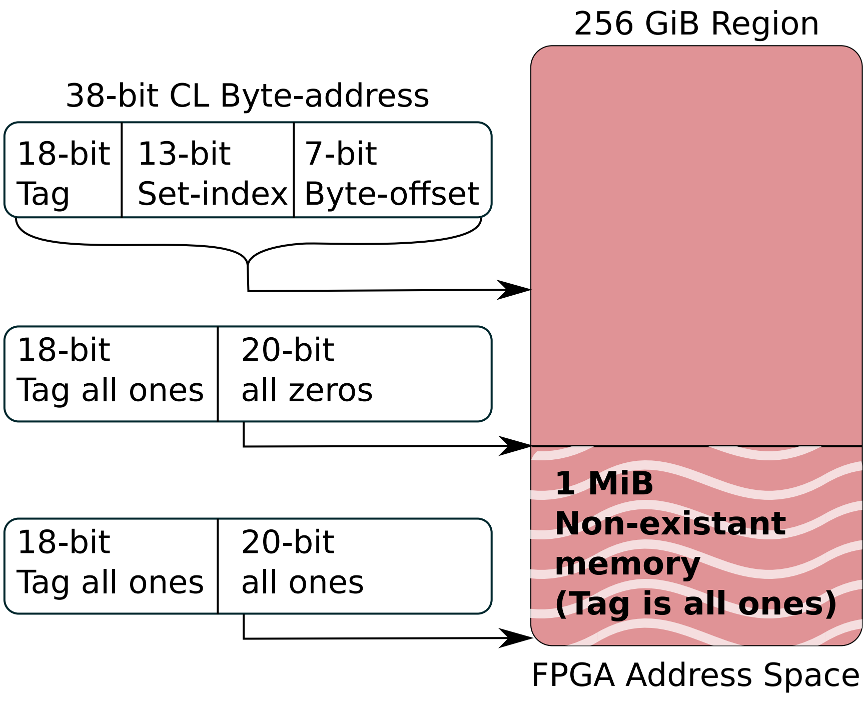


Figure 8.6: *Of the 256 GiB of memory exposed by DC, 1 MiB is non-existent and is not usable. The non-existent region is identified by cache line address where the tag field is all ones. This is done to optimize performance of DC.*

8.3.5 DC architecture top level

The reason for organizing DCUs into two DCSs is to take advantage of the inherent odd, even channel parallelism provided by ECI. It only makes sense to extend it further into the memory channels as well. Each DCS exposes an AXI interface that access mutually independent (odd and even) regions of the FPGA address space.

Putting it all together, the overview of the DC architecture is as shown in Figure 8.7. The DC has two DCSs. Odd numbered VCs (both ECI and local) channels are connected to one DCS and even numbered VCs are connected to the other. Each DCS exposes an

AXI bus to access memory. The two AXI channels are mutually independent. For sake of simplicity when depicting the DC, the two AXI channels are represented as one as shown in Figure 8.1. Next we will look into the architecture of the DCS.

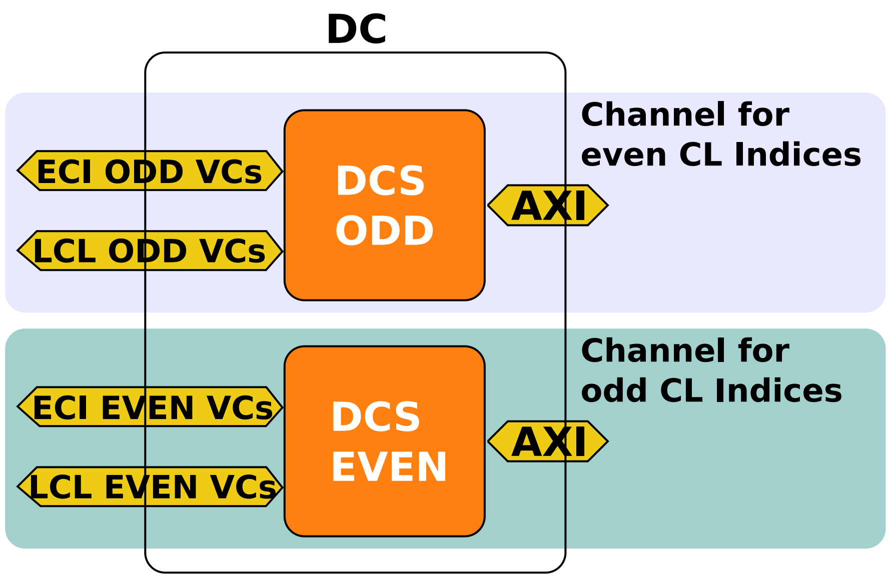


Figure 8.7: *DC architecture top level: The DC consists of two parallel DCSs units.*

8.4 Directory Controller Slice (DCS) Architecture

8.4.1 DCS interface

The DC comprises of two DCSs where each DCS has a number of DCUs. Each DCU has a unique DCU-ID and within a DCS, each DCU is identified by its DCU-IDX as shown in Figure 8.5. The DCS receives coherence messages of different types from multiple sources. Each coherence message is associated with a cache line address, and part of the cache line address is the ID of the DCU responsible for handling it.

The different types of interfaces exposed by a DCS to receive coherence messages are as follows: An interface to connect to ECI to exchange coherence messages with the CPU, a second interface that exposes read, write descriptor interface to be converted to AXI and connected to the memory, and finally a local interface for applications on the FPGA to interact with the coherence protocol through DCS. Each interface is made up of one or more channels with valid-ready flow control. These interfaces are shown in Figure 8.8.

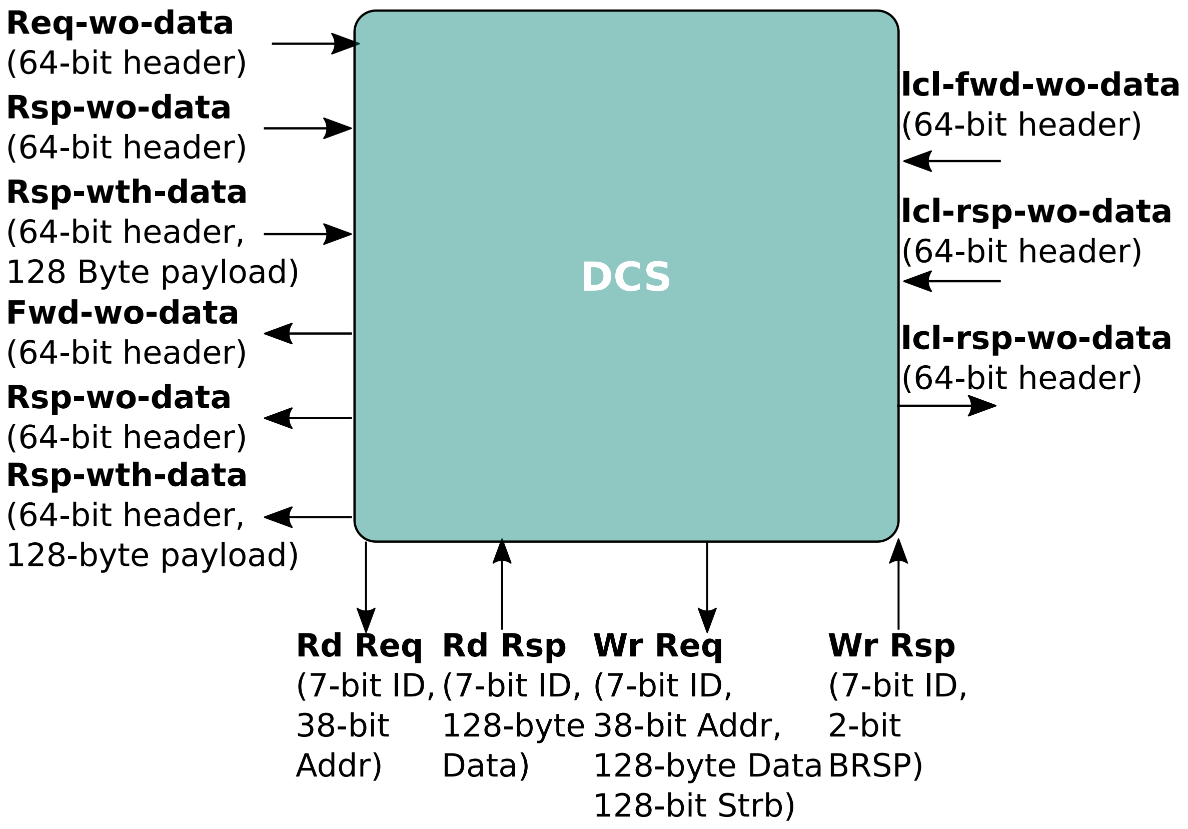


Figure 8.8: *ECI, Local and Memory interfaces to DCS: All interface channels are independent of each other and have valid-ready flow control.*

The DCS interface to ECI consists of three incoming and three outgoing channels. The incoming channels are request-without-data, response-without-data and response-with-data-channels. The request-without-data channel receives upgrade requests from the CPU and both response channels receive voluntary downgrades as well as forward-downgrade acknowledgments from the CPU. In contrast, messages are sent to the CPU via outgoing channels namely forward-without-data, response-without-data and response-with-data.The forward channel is used to issue forward downgrade requests to the CPU. The response channels are used to issue responses to upgrade requests from the CPU.

In general, all ECI channels have a 64-bit header (of which the cache line address is a part). Each channel is also assigned a VC number (not shown in figure) to distinguish between them. Channels with data additionally have up-to 128-bytes of payload. The size of the payload can vary between 1 and 4 sub-cache-lines (each sub-cache-line being 32 bytes), and is indicated by the *dmask*-field in its header. The dmask-field also identifies the exact byte-address to which the sub-cache-line is to be written to. This allows for data that are smaller than a cache line to be compressed, thereby reducing the amount of data flowing through the interconnect.

The read descriptor interface to memory has an outgoing read request channel and incoming read response channel. The read request channel is used to initiate read transactions by issuing the cache line byte-address to read along and tagging it with a transaction identifier. An open read transaction is completed when response data, tagged with the transaction identifier, is received through the read response channel.

The write descriptor interface also has outgoing write request and incoming write response channels. The write request channel initiates a write transaction by tagging it with a transaction identifier and issuing the cache line byte-address to write to along with write data and byte-enable strobe signals. Similar to read transactions, write transactions are also terminated with a matching transaction identifier is received in the write response channel along with information indicating if the write completed successfully.

The memory descriptor channels have 7-bit transaction ID and a data-bus that is cache line size wide. Additional modules are available to convert these descriptor interfaces to AXI signals. It is to be noted that in the current design, write requests are at the granularity of sub-cache-lines whereas reads are always at the granularity full of cache lines. This is because reading sub-cache-lines is only an optimization which we decided not to implement.

Design Choice 8.6. *The DC supports sub-cache-line granularity write requests to memory, but read requests are always at the granularity of a cache line. The only impact of this design choice is reduction in performance.*

Finally, the local interface has an incoming forward-without-data channel for the FPGA application to issue clean or clean-invalidate requests, an outgoing response-without-data channel containing acknowledgments for previously issued clean, clean-invalidate requests, and an incoming response-without-data channel for the FPGA application to unlock previously locked cache lines (if allowed by the protocol state machine). All channels have a 64-bit header with opcode indicating the type of operation and the cache line address embedded into it. The local channels also have distinct VC numbers associated with each channel. There are no data channels in the current iteration of the local interface.

8.4.2 DCS control and data-paths

In order to simplify routing of coherence and memory events to the numerous DCUs within a DCS, the information from the various incoming channels are split into control and data. Control information in a channel is any information except cache line data. This includes the 64-bit headers in ECI and local interfaces, as well as read/write response transaction IDs from the memory interface. The *control-path* routes control information to the DCUs where as *data-path* can be used to store and retrieve data as needed. The control and data-paths are shown in Figure 8.9.

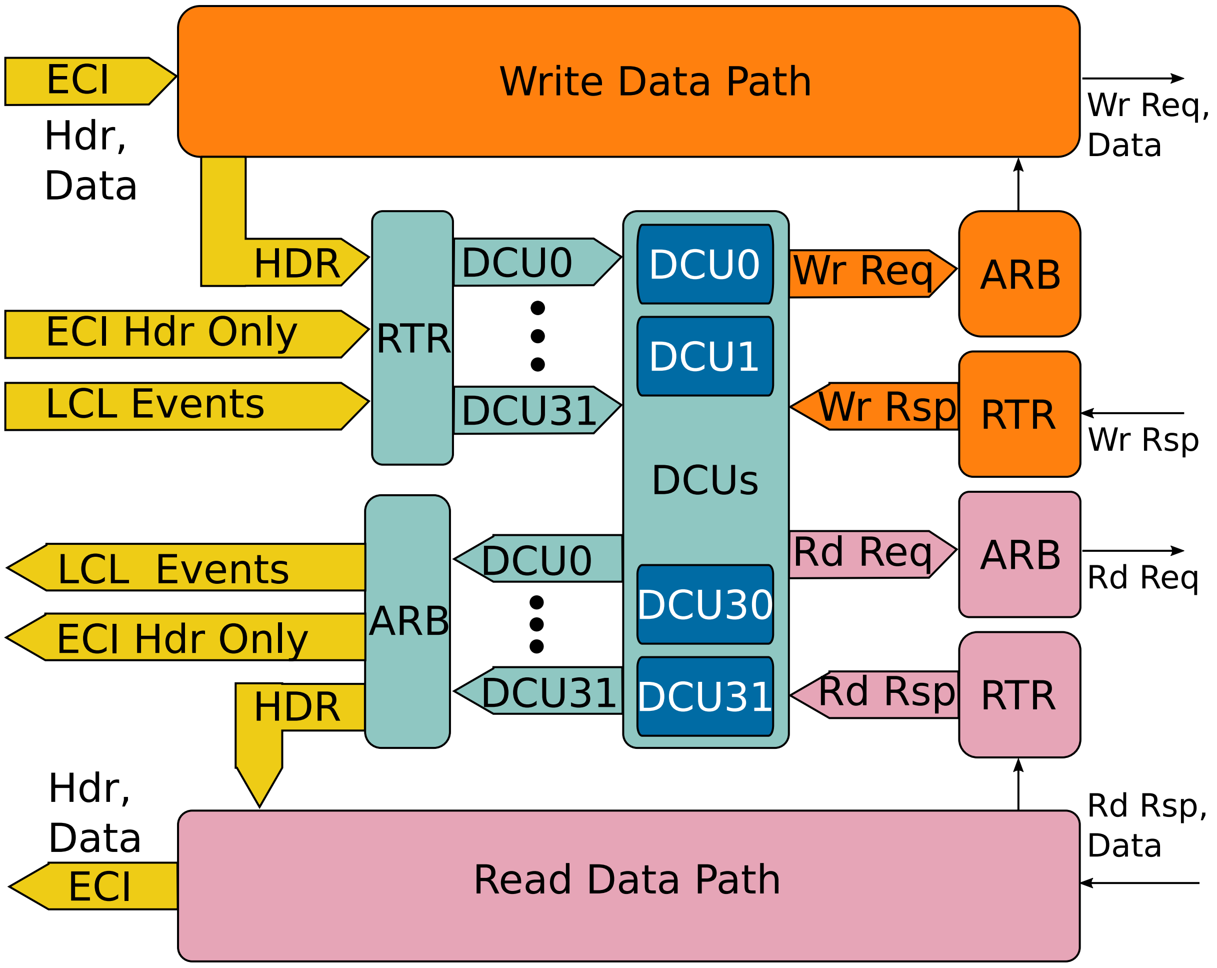


Figure 8.9: *DCS control and data-paths: Control information is routed to the DCUs that lie in the control-path and cache line data is stored and retrieved from data-path. The data-paths are separated for read and write operations to avoid deadlocks.*

The DCS has two data-paths, the write-data-path to hold data that is to be written to the memory and the read-data-path to hold data read from the memory. The reason to separate out read and write data-paths is that read and write operations are independent of each other and having a single data-path for both operations would create dependencies between the operations which can lead to deadlock. Each DCU in a DCS is *direct-mapped* to one slot in the data-path buffer (indexed by the DCU-IDX) to store and retrieve cache line data.

Design Choice 8.7. *The DCS has separate control and data-paths to avoid routing cache line data to the DCUs. Only control information is routed to and from the DCUs and each DCU in a DCS is direct-mapped to a slot in the data-path buffer to store and retrieve cache line data. Separate data-paths are available for read and write DC operations to avoid deadlocks. The alternative would be to route data along with control information to the DCUs which would cause routing congestion.*

On the write-data-path, we allow only responses (with data) from the CPU to be written to the memory through the DCS. This means that if an FPGA application wants to write to the memory, it has to interact with the DCS through its local interface channels and have its own data-path outside the DCS. This design choice is made to modularize the DCS, as not all FPGA applications would require writing to memory. The same goes for FPGA applications reading from memory. If, in the future, the DCS has to support reads and writes through the local interface, add additional data-path channels and do not mix with the existing data path channels as this might lead to deadlocks due to circular dependencies.

When dirty data from CPU arrives in the response-with-data ECI VC, it gets written in the write-data-path buffer at a slot that is identified by the DCU-IDX in its cache line address, provided the slot is empty. If the slot is not empty, both header and data in the response-with-data ECI VC gets blocked till the previously stored data is retrieved and the slot becomes free. Once data is stored, the header is routed to the DCU (again using the DCU-IDX) through the control-path. The DCU would eventually issue write request control-signals. The write request would be tagged with the DCU-ID as the transaction ID, and write strobe signals are generated from the dmask-field in the header. The stored data is then retrieved by indexing the buffer with DCU-IDX from the transaction ID, before issuing a write request to memory.

On the read-data-path, the read response is stored in the data-path buffer. Similar to write transactions, read transactions are also tagged with the DCU-ID. Thus the transaction

ID from the read response is used to identify the index to store data. Once read response data is stored, the control signals (transaction ID in this case) is routed to the appropriate DCU. Eventually the DCU generates a 64-bit header (as a response to what caused the read request), the cache line address of which is used to retrieve the data before sending it through an outgoing data channel. In the current design, the read-data-path is used for serving upgrade requests from the CPU. FPGA applications that require reading the memory must interact with the DCS through the local interface and have its own data-path outside the DCS.

Architecture of control-path: As shown in Figure 8.9, each incoming interface channel has routing logic to route control signals to the DCUs. Control signals output from DCUs are arbitrated to choose one set of signals for an outgoing interface channel. The arbitration for each outgoing channel is independent of other outgoing channels. For example, the arbitrated read and write requests can be from different DCUs.

Architecture of data-path: In general, the data-path has to ensure that stored data is not overwritten before it is retrieved. The data-path achieves this by blocking any writes to the buffer where this rule is violated. Since each slot corresponds to a single DCU, this also means that each DCU can have only one outstanding operation on a data-path.

The read and write data-paths are similar in all cases except that writes can happen in sub-cache-line granularity and reads are always cache line granularity. Incoming data from ECI response-with-data channel into the write-data-path can be compressed and would have to be decompressed using the dmask-field i.e. the sub-cache-lines would have to be mapped to the correct byte-address they should be written to. This operation is not required in the read-data-path.

The architecture of a generic data-path is shown in Figure 8.10. Control and data are passed into the gate-keeping module (DP_GATE) which ensures that a DCU has only one ongoing operation in the data-path. When a slot is available, the gate-keeping module sends the control signals to the DCUs and writes the data into a data-store. A module that maps compressed ECI data to its byte-address (based on dmask-field) can be optionally instantiated within the gate-keeping module.

The decompressed cache line size (1024-bit) data is then split into two chunks of 512-bits each by a serializer module (DP_WR_SER) before writing into the data-store (DP_STORE). This is done to reduce the size of the bus in order to ease routing and reduce congestion.

Depending on the number of DCUs in a DCS, the data-store has 16/32/64 cache line

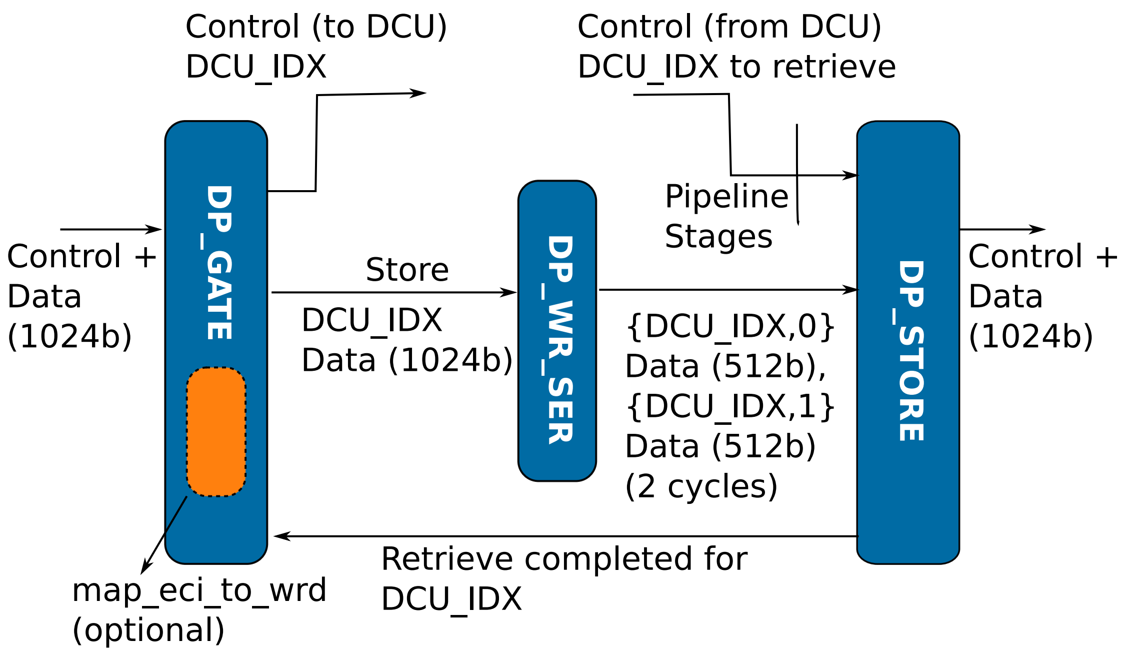


Figure 8.10: *DCS data-path architecture: Each DCU in a DCS is direct-mapped to a slot in DP_STORE to store and retrieve data. The DP_GATE module prevents overwriting the contents of a slot before it is retrieved. It also splits the incoming coherent message into control and data. The 1024-bit data is serialized into two chunks of 512-bits by DP_WR_SER before being stored in DP_STORE. Eventually this data gets retrieved and issued to the output. Pipeline stages at retrieve input avoids race conditions where retrieve can happen before store.*

sized slots.The data-store uses the DCU-IDX to store cache line sized data in 2 cycles and retrieve the same in 1 cycle. Since storing data happens in 2 cycles and retrieving data happens in 1 cycle, there are pipeline registers present at the retrieve interface to ensure data is stored before retrieval. Upon retrieval, the DCU-IDX is sent by the data-store to the gate-keeping module indicating that the slot for the DCU has been freed.

8.4.3 Memory descriptor interface to AXI interface

Figure 8.11 shows the modules that are used to convert the memory descriptor interface of DCS to a standard AXI primary interface. It is up to the FPGA application developer to choose which interface to work with.

The AXI primary interface consists of five parallel channels with valid-ready flow control.

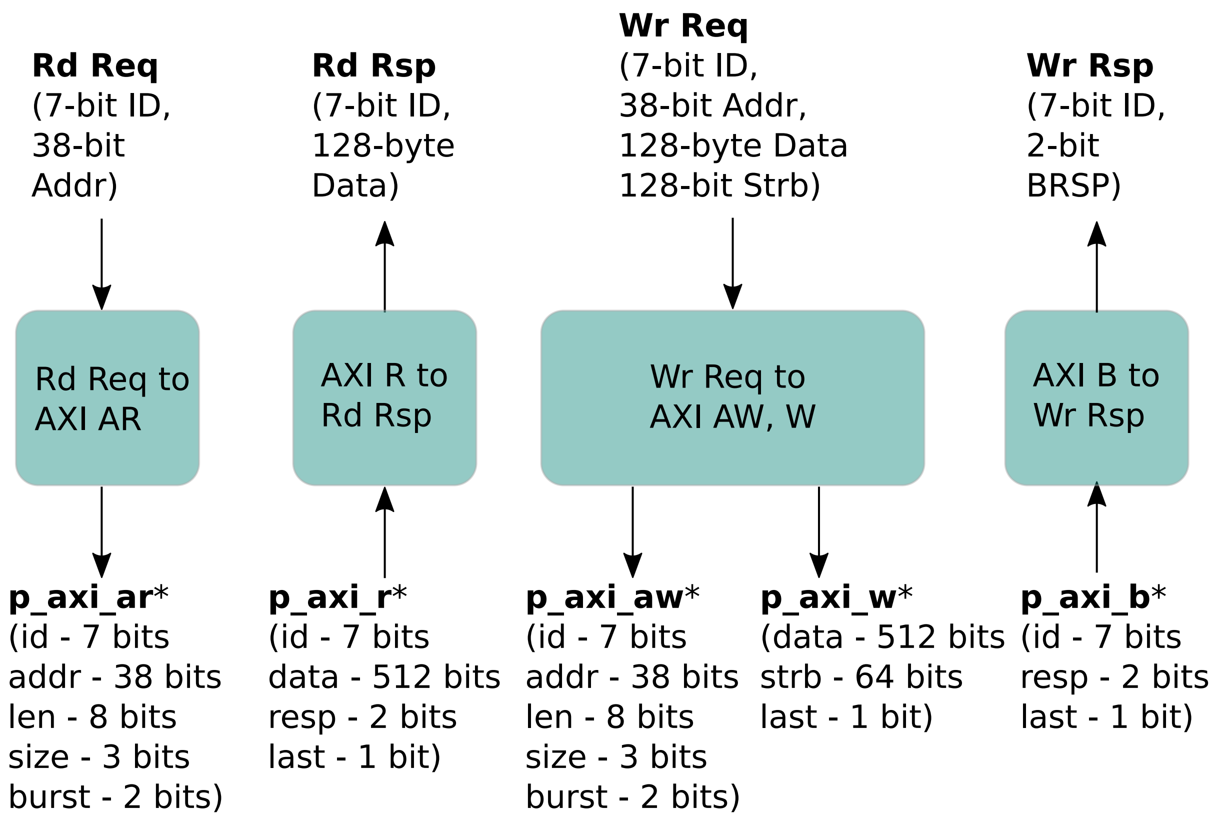

Rd Req (7-bit ID, 38-bit Addr)
Rd Rsp (7-bit ID, 128-byte Data)
Wr Req (7-bit ID, 38-bit Addr, 128-byte Data 128-bit Strb)
Wr Rsp (7-bit ID, 2-bit BRSP)
Rd Req to AXI AR
AXI R to Rd Rsp
Wr Req to AXI AW, W
AXI B to Wr Rsp
p_axi_ar* (id - 7 bits addr - 38 bits len - 8 bits size - 3 bits burst - 2 bits)
p_axi_r* (id - 7 bits data - 512 bits resp - 2 bits last - 1 bit)
p_axi_aw* (id - 7 bits addr - 38 bits len - 8 bits size - 3 bits burst - 2 bits)
p_axi_w* (data - 512 bits strb - 64 bits last - 1 bit)
p_axi_b* (id - 7 bits resp - 2 bits last - 1 bit)

Figure 8.11: *DCS to AXI primary interface: The AXI primary interface has 5 separate channels address-read (AR), read-data (R), address-write (AW), write-data (W), and write-bresp (B). It has 7-bit transaction ID and 512-bit data-bus which is different from 1024-bit data-bus of the descriptor interfaces.*

The channels are address-read (AR), read-data (R), address-write (AW), write-data (W) and write-bresp (B). The address-read channel issues the address to read from and read-data channel carries the return data back. The address-write and write-data channels have data to be written and write-bresp channel indicates when write has completed. Requests will be tagged with a 7-bit transaction ID and the response for a request will be tagged with the same 7-bit transaction ID.

It is to be noted that the data-bus at the AXI interface is 512-bits to match the data-bus of the DRAM Memory Interface Generators [Xil22a] (aka replica). It is the responsibility of the modules to bridge between the 1024-bit memory descriptor bus and the 512-bit AXI bus. Thus on the AXI but there will be two beats in a burst with 512-bits (64-bytes) per beat and the burst type would always be *INCR* to indicate that the replica should increment address after each beat.

8.4.4 Saturating ECI transmit bandwidth

In the scenario where there is no coordination required between cache lines, we have seen that the read and write descriptor channels output from the DCS would be connected to a DDR memory channel. In this section, we will calculate the number of outstanding memory (read and write) transactions that would have to be issued by the DCS to saturate ECI transmit bandwidth.

The ECI transmit bandwidth is used to send cache line data (128 bytes) to the CPU by reading from DDR memory. Let us assume that the ECI transmit bandwidth is around 16 GiB/s (ballpark from 2 socket ThunderX-1 system) and the memory channels have up to 36 GiB/s (2 memory channels with each 18 GiB/s). Of the two, ECI channel is the bottleneck when reading data from DDR and sending it to the CPU. Thus we need to have enough outstanding read and write transactions to saturate ECI bandwidth to avoid unnecessary queuing.

In order to do this, we need to calculate the bandwidth-delay product of the ECI channel. Let us assume that the bandwidth at the output of a single DCS is half the ECI bandwidth, say 8 GiB/s. The round-trip latency of DDR channels is around 70ns and since often times we would want an application between the DCS and DDR, we conservatively assume the latency of this application and DDR channels is around 300ns.

The amount of data transferred per memory request is the size of a cache line, 128 (2^7) bytes. Thus the number of requests required per second to saturate the 8 (2^{33}) GiB/s bandwidth would be $\frac{2^{33}}{2^7} = 2^{26} requests/second$.

The bandwidth-delay product would give the number of outstanding requests that would have to be issued.

$$\begin{aligned} Nr\ of\ outstanding\ requests &= 2^{26} * 300 * 10^{-9} \quad (Bandwidth - delay\ product) \\ &\approx 20 \end{aligned} \tag{8.1}$$

Thus with 16/32/64 (powers of 2 close to 20) outstanding memory requests from a DCS we can expect to saturate ECI bandwidth. This number would also indicate the depth of the data-store per DCS data-path. Since read and write channels are independent, each channel should individually have 16/32/64 outstanding memory requests.

8.4.5 Saturating ECI receive bandwidth

In addition to memory transactions, the DCS can also initiate forward downgrade transactions on FPGA-homed cache lines(chapter 6). Assuming each forward downgrade request brings back a cache line worth of data on the receive channels, we should have enough number of outstanding ECI transactions to saturate the receive bandwidth.

The round-trip latency of ECI channel is around 230ns as measured in Figure 9.2. If we round this off to 300ns, using the same calculation in equation 8.1, we can also saturate the ECI bandwidth if we have 20 outstanding forward downgrade transactions per DCS.

Thus with 16/32/64 outstanding forward downgrade ECI transactions that are outstanding per DCS, we can saturate ECI bandwidth as well.

Design Choice 8.8. *Based on bandwidth-delay product we need to have 16/32/64 outstanding memory and forward-downgrade transactions per DCS in order to saturate ECI transmit and receive bandwidth. This number also indicates the depth (capacity) of data-store for a single DCS data-path.*

8.5 Directory Controller Unit (DCU) Architecture

8.5.1 Interface and basic operation

The DCU is responsible for providing coherent access to a pre-defined subset of the FPGA address space. The DCU sits in the control-path of the DCS and deals with coherence transaction headers and memory transaction control signals. It also determines the order in which these events are handled. Each DCU implements a portion of the directory which stores the present state of a cache line. It also implements the coherence protocol which takes a coherence event and present state of a cache line to specify the next state and action to be performed. The DCU should provide the following guarantees: First, the DCU provide a separate state machine for each cache line to keep coherence transactions of cache lines separate. Second, the state transitions on cache lines should be faithful to the transitions that is dictated by the DC protocol state machine in order to replicate all guarantees provided by the state machine. Third, the DCU should be deadlock free when managing its limited resources. Fourth, the DCU should have reasonable performance.

In addition to these guarantees, the DCU should be modular enough to accommodate changes to the protocol as well as FPGA application interface easily.

Performance of the DCS is achieved by running multiple DCUs in parallel. Hence the design of a DCU is simple, un-pipelined and optimized for resources. The basic operation of a DCU is as follows: Of the multiple incoming channels, the DCU chooses one event to handle at a time. The event gets decoded to identify the type of event and the cache line address. The address is then looked up in the directory to get the present state of the cache line. The event type and present state are looked up on to the coherence protocol table which provides the next state and action to be performed. The DCU finally performs the action and updates the directory with the new state. There are six types of actions performed by the DCU: **no-action** (just updating the directory), initiating **read** transactions with memory, initiating **write** transactions with memory, initiating **forward-downgrade** transactions with CPU, issuing **responses headers** to CPU/FPGA initiated coherence transactions, and **delaying** an incoming coherence header to be retried later. When the event completes the basic operation and is not delayed for later, the event is said to be handled.

Events can be delayed (to be retried later) either by the DC protocol state machine or when the DCU does not have resources available to handle the event. It is to be noted that when an event is delayed, the next state that gets written into the directory is the same as its present state. In other words, the state of the cache line in the directory does not change when an event is not handled.

Note 8.3. *We have seen earlier that the protocol state machine can stall an event to be handled later (see design-choice 5.2). This delaying of an event by the protocol state machine is different from delaying of event by the DC. The DC has to delay an event whenever the protocol state machine calls for it. In addition the DC can delay an event when there is not enough resources. The protocol state machine only delays coherence requests to avoid deadlocks but the DC can delay event of any type. For example, a voluntary downgrade with data event can be delayed by the DC because the write channel is busy. The DC should guarantee that such delays are temporary and would be resolved when the resource becomes available.*

The interface to the DCU is shown in Figure 8.12. The DCU has separate request-response interfaces for both memory read and write transactions. Only control information is generated in these channels which will be used to retrieve the actual data. In addition

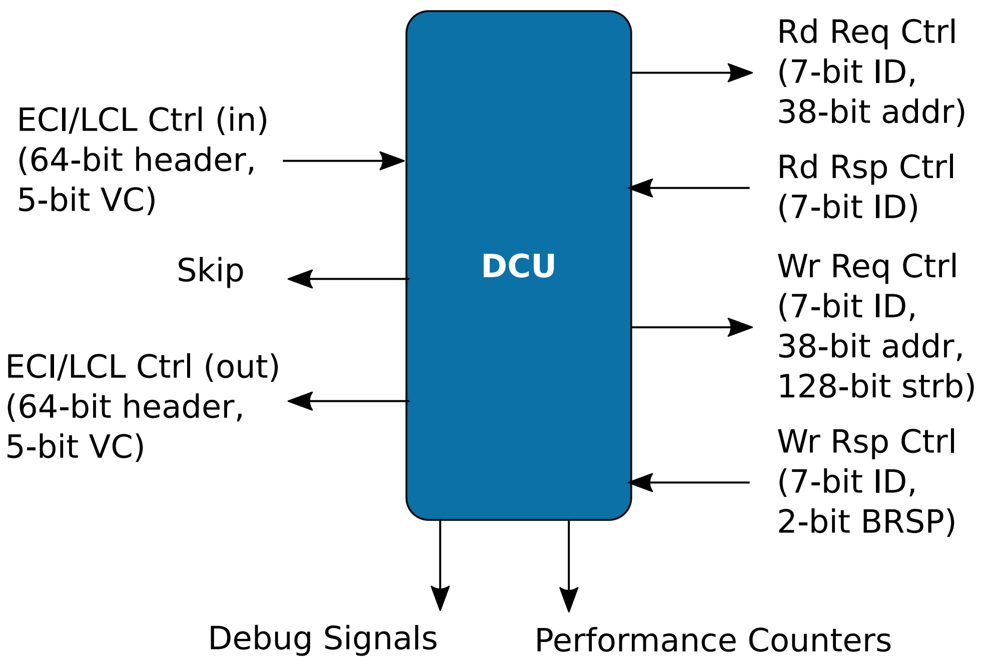


Figure 8.12: *DCU interface: One incoming channel for all ECI and local VCs, one outgoing channel to all VCs. For memory read transactions, separate read request and response channels with only control information. Similarly for memory write transactions, two separate write request and response control channels. Finally, a skip signal to delay an incoming coherence event to try again later.*

to the memory control interface, the DCU has *one* incoming and one outgoing channel to exchange coherence headers with all VCs (both ECI and local). There is only one incoming VC interface in DCU for all coherence events because the protocol state machine handles only one coherence event at a time (design-choice 4.2). Finally, it has a *skip* signal that indicates whenever an incoming ECI or local event is to be delayed for later. Each channel has a valid-ready flow control. Note, the DCU here receives only control information and no data.

8.5.2 Design considerations

Each cache line has a separate state machine: It is to be noted that the basic operation of the DCU (discussed in subsection 8.5.1) on one cache line does not affect the state of any other cache line. Handling a coherence event on a cache line only updates the state of that cache line in DCU's directory. Furthermore, the DCU maintains a unique state for each cache line in its directory. This ensures that coherence events (and transactions) on one cache line does not affect the state of a different cache line within the DCU and each cache line has a separate state machine.

Faithfulness of DCU protocol implementation to DC protocol state machine: Since the DCU is un-pipelined, its basic operation for a coherence event is *atomic*: All steps in the basic operation are completed for an event before the next event is chosen. With the basic operation being atomic, we can be guaranteed that the state transition in the DCU will be faithful to the transitions indicated by the DC protocol state machine: When an event is handled, the next state as dictated by the protocol is updated in the directory before a next event is chosen, and when the event is not handled, its state in the directory does not change.

Pipelining the design would consume additional resources that are required to handle hazards due to control dependencies when multiple events for the same cache line are being handled simultaneously in the pipeline. For example in a pipelined design, the next state that would eventually be written back to the directory should be forwarded to a previous pipelined stage to choose as present state instead of the stale contents of the directory.

Design Choice 8.9. *Each DCU is un-pipelined and optimized for resource consumption. Having an un-pipelined DCU avoids hazards that can arise due to control dependencies and guarantee that the implementation of the DC protocol is faithful to the protocol state machine.*

Resource deadlock avoidance: The DCU has finite resources and has to deal with resource contention. The directory and buffers in memory hierarchy are examples of finite resources available to the DCU. As such there is a possibility that a DCU is unable to handle the chosen event at that point in time. Since resources get freed only when DCU handles coherence events, deadlocks can occur if the DCU stalls waiting for a resource. For example, consider the scenario where a DCU wants to issue a read request but the memory is not ready to accept new requests. If the DCU stalls waiting for the memory to accept its request, it will not be able to sink read responses that might free up memory resources to handle new requests. *Thus the DCU should never stall and be able to skip an event that cannot be handled, to try a different event.* Furthermore, the last message in a chain of dependencies, the response events in this case, always free up internal resources and retire transactions. For example, handling a response to a memory request allows the

DCU to initiate a new memory transaction. *It is the responsibility of the coherence protocol to always sink response events and the DCU prioritizes response channels to aggressively free resources.* This is indeed the case as seen in design-choice 5.3.

Design Choice 8.10. *The DCU should never stall waiting for a resource. Whenever a resource is not available, the DCU should be able to delay the event and try to handle a different event. This allows the DCU to handle events that can free up internal resources and avoid deadlocks arising due to resource contention. These delays are temporary and the DCU should eventually retry the delayed event to handle it when resources get freed.*

The DCU also prioritizes events in response channels over requests to aggressively free up internal resources although this is not necessary to avoid deadlocks.

As will be seen in subsection 8.5.4, resource contention in the DCU can happen either at the directory or at any of the outgoing valid-ready channels shown in Figure 8.12. Thus the DCU should never stall waiting for the directory to free up or for handshake to happen at any of the outgoing valid-ready channels.

That said, the current DCU can potentially deadlock due to two incorrect assumptions that were made: Stalling when unable to issue an outgoing coherence header and having only one outgoing channel for all VCs. Stalling is an obvious reason for deadlocks as described in the previous section. The current version of DCU stalls when a handshake has not occurred in the outgoing channel that carries the coherence header response. Having only one outgoing channel for all VCs can lead to headers of different classes stuck behind each other. Future design changes should reconsider the assumptions. Although, due to the CPU's resilient credit-based flow control (the CPU also always sinks response messages from the DC), we have never run into a deadlock (even under heavy loads) in practice.

Takeaway 8.1. *Future version of DCU should not stall if the DCU is unable to issue coherence messages to the CPU (i.e. if the corresponding VC is not ready). Future version should also consider having separate channels for different outgoing coherence messages based on its VC instead of having only one out-going channel for coherence messages as present in current DCU.*

Saturating ECI bandwidth: Each DCU can initiate both ECI and memory transactions. For example, forward downgrade transactions can be initiated by the DCU on ECI, and memory read and write transactions can be generated with the memory. We have seen in design-choice 8.8 that have 16/32/64 outstanding read and write and forward downgrade transactions per DCS can saturate memory and ECI bandwidth. Since we want each DCU to be optimized for resources, we allow each DCU to have one outstanding transaction in each of read, write and forward-downgrade operations which serializes these transactions. It is to be noted that having an on-going, say, read transaction does not prevent the DCU from issuing a write or forward-downgrade transaction. Thus to saturate the ECI bandwidth, we need at least 16/32/64 DCUs per DCS where each DCU has one outstanding transaction.

Design Choice 8.11. *Each DCS should have 16/32/64 outstanding read, write, and ECI transactions in order to saturate ECI bandwidth. The design choice is made to limit each DCU have one outstanding transaction of each kind and to have 16/32/64 parallel DCUs per DCS. Thus each DCU is blocking.*

Need for modular DCU design: The DCU also implements the coherence protocol that, given a coherence event and present-state of a cache line, determines the next state and action to be performed. Although the CPU implements a full MOESI protocol, the DCU is free to implement a subset of the protocol that is suitable to its needs. For example, the *owned* (O) state is required for inter-cache transfers and since there is no cache on the FPGA the DCU can implement a MESI variant and still operate seamlessly with the CPU's protocol. Furthermore, the application requirements can be used to fine-tune the protocol. For instance, in an application where only the CPU accesses FPGA memory, the DCU does not have to support interactions with coherence controllers on the FPGA. Thus sevaral variants of the protocol are possible and it is beneficial to have a modular architecture that accomodates this.

8.5.3 DCU interface with DCS

In this section, we will look at how the DCU interfaces with the DCS. As seen in Figure 8.9, the routing logic in DCS routes control information from a number of VCs and memory channels to the DCUs based on cache line index. The interface to the DCU is shown in Figure 8.12. In addition to the memory control interface, it has one incoming and one outgoing channel to exchange coherence headers with all VCs (both ECI and local).

Since there is only one incoming interface for all VCs in the DCU, arbitration is required between various VCs of the DCS to select one. Once a VC is chosen, the DCU peeks at the header at the top of the channel to determine if it can be handled. The DCU would then pop the header off the VC if it can be currently handled, or switch to a different VC if not. This arbitration is performed by a round-robin arbiter as shown in Figure 8.13, with a special skip signal to switch VCs without popping. It is to be noted that skipping a VC would cause head-of-line blocking in the skipped channel which might affect performance.

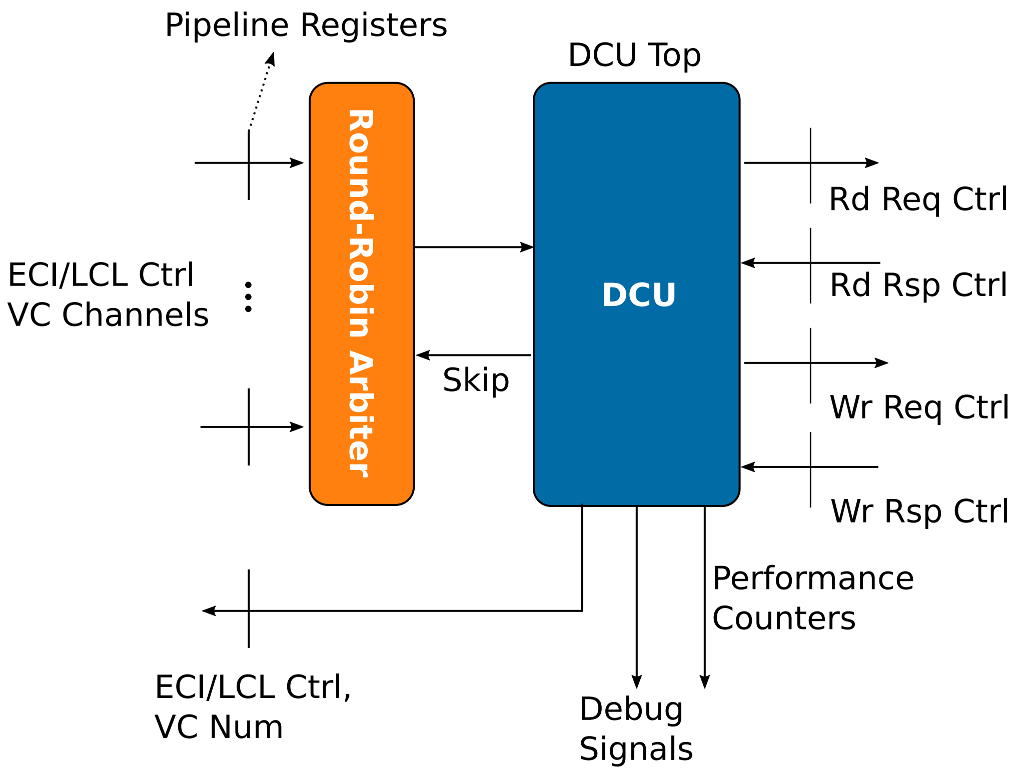


Figure 8.13: *DCU interface to DCS: Incoming coherence messages from ECI and local VCs are arbitrated to choose one coherence message at a time. This is done because the protocol state machine itself handles only one coherence event at a time.*

On the outgoing VC interface, the DCU issues coherence headers (no data), with the VC number determining the exact VC to which the header has to be routed to. Finally memory transactions are initiated and retired through the memory control interface.

It was seen earlier that each DCU is blocking and can have at most one outstanding read, write, and forward downgrade ECI transaction. All these transactions are tagged with the DCU-ID to facilitate routing of responses.

All incoming and outgoing channels are registered using pipeline stages. These buffers isolate the DCU and reduce head-of-line blocking on the DCS VCs by holding coherence headers as they are being handled.

Design Choice 8.12. *All memory and ECI transactions that are initiated by a DCU will be tagged with its DCU-ID. This is done to facilitate routing of responses to the correct DCU.*

8.5.4 Design of DCU

The basic operation of DCU described in subsection 8.5.1 gives us an idea of the components required in the DCU: an event decoder, a directory to store and retrieve cache line state information, a protocol look-up table, individual components for each action to be performed and a controller that orchestrates everything. These components are shown in Figure 8.14.

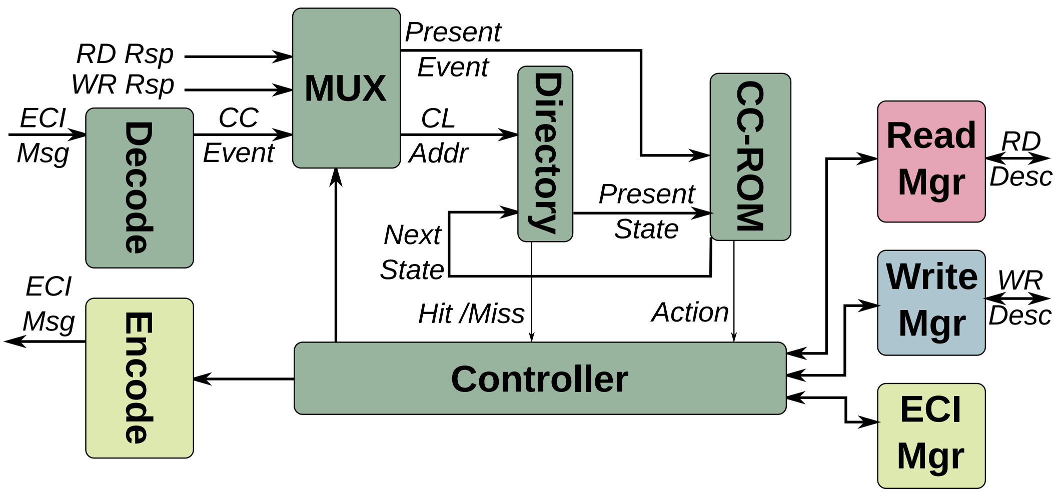


Figure 8.14: *DCU architecture: Incoming coherence event is decoded to get the event and cache line address. Then the present state of the cache line is looked up in the directory. The present state and event is then looked up in the protocol ROM to get the next state and action. Once the action is performed the state of the cache line gets updated in the directory. The design is not pipelined and all these operations are guaranteed to be atomic.*

The event decoder takes in the coherence header and uses the opcode and VC number to identify the coherence event and uses this information to extract its bit-fields including the transaction ID, dmask and cache line address.

In addition to the decoded coherence message, there can be response events from memory transactions waiting to be processed. The DCU prioritizes these responses over the coherence message (for reasons discussed in subsection 8.5.2) to select one coherence event and pass its cache line address to the directory.

The set and tag information from the chosen cache line address is passed to the directory (also called Tag-State-Unit (TSU)). The set information is used to index into the directory and the tag information is compared with tags in the ways to check if there is a match. If a match is found, the directory retrieves the present-state of the cache line along with the way in which the hit occurs. Not finding a matching tag means the cache line is not cached anywhere in the system and directory returns the present-state as *Invalid*(I). The way information returned by the directory in the case of a hit, would be used later when updating the state of the cache line in the directory.

The present-state of the cache line and the chosen coherence event are then looked up the coherence protocol table (Cache Coherence Protocol Read Only Memory (CC-ROM)) to get the next-state and action to be performed. The action is then coordinated by the controller. Having the coherence protocol as a look-up table allows us to switch coherence protocols without changing the rest of the design.

For actions that require initiating a new transaction, the controller invokes one of three Transaction Managers (TMs): read TM, write TM and ECI TM. Each TM stores the original coherence header that caused the transaction and initiates a single outstanding transaction. The TMs also indicates to the controller when it is busy so the controller can delay all events that require this resource. Once the transaction is retired (response is received), the TM retrieves the stored header, generates a completion event and is now ready to issue a new transaction. The completion event is then eventually handled by the DCU as dictated by the coherence protocol. It is to be noted, that all transactions initiated by the TMs are always tagged with the DCU-ID to facilitate routing of the remaining chain of coherence events. This also helps decouple coherence transactions from memory transactions as they have different IDs.

Coherence headers have to be issued when initiating a forward-downgrade or responding to a previously initiated coherence transaction. These headers are generated by an encoder module enabled by the controller. The controller identifies the exact coherence message to be sent based on the action prescribed by the coherence protocol. It also provides information required to generate the header such as coherence transaction ID and cache line address. The encoder generates the header along with the VC number and issues it via the outgoing channel.

Finally, for all actions except delaying a coherence event, the directory gets updated with the next-state defined by the coherence protocol. The directory gets updated with the present-state when an event is to be retried later.

Design Choice 8.13. *By implementing the protocol state machine as a read-only lookup table, we can easily swap it out for a different variant of the DC protocol. Additionally, adding new VCs for messages on the local interface does not require changes in the DC. The only changes that might be required are for decoding and encoding new local coherence messages.*

Resource contention in this DCU design can arise at the directory with its limited storage for cache lines tags and states, or at the transaction managers which allow only one on-going transaction. The DCU should not not stall whenever any of these resources are busy to avoid deadlocks (design-choice 8.10).

8.5.5 Protocol scenarios and DCU pathways

In this section, we will look at the different pathways in the DCU to handle coherence events. The scenarios described here are to illustrate how the DCU behaves, and to this end, we assume that the non-locking variant of the coherence protocol is programmed into the CC-ROM. For all these scenarios we have the following steps in common. The DCU peeks at the chosen coherence event, decodes it, obtains the cache line's present state from the directory and, looks up the CC-ROM to get the next state and action to be performed. The coherence event is not popped off it's VC (not handled yet). We will look at how the DCU executes these actions.

The simplest scenario is for voluntary downgrade responses without data. The DCU does not have to perform any action other than updating the directory with the next state. In other words, the action from CC-ROM is *no-action*. Once the directory is updated, the coherence event is popped off its VC and is considered to be handled.

Slightly more complex are voluntary downgrade responses with data. The dirty data would have to be written back to the memory (CC-ROM action is *write*). In this case, the controller on the DCU checks if the write TM is busy. If the write TM is busy, the DCU delays the coherence event and chooses a different event to try.

Note 8.4. *For the curious reader who is wondering why the DC delays (or stalls) a response coherence event when the protocol state machine explicitly forbids it (design-*

choice 5.3), it is to be noted that this delay is temporary as the write TM is guaranteed to be freed up when the write response is received at which point the delayed coherence event would be resumed.

If the TM is free, the controller passes on the coherence event along with relevant information to the TM, updates the directory with the next state and, marks the coherence event as handled by popping it off its VC. The DCU is then free to handle other coherence events. The write TM accepts the information from the DCU controller and sets a busy flag. It stores the coherence event and initiates a write transaction. The write TM is busy for the entire duration of the write transaction till the write response arrives. Once the write response arrives at the TM, it retrieves the stored coherence event and sends it along with control information from the write response to the DCU and waits for it to accept this information. The TM would become free, i.e. the write transaction is retired, when the DCU accepts this information.

The third scenario we see is what the DCU does when it receives a write response information from the write TM. In this case, the action might be as simple as accept the information sent by the TM and update the directory (CC-ROM action is *no-action*).

In the fourth scenario we discuss what the DCU does when it receives an upgrade request. The DCU controller invokes he read TM to initiate a read transaction. The read TM behaves exactly like the write TM with the only difference being that a read transaction is initiated instead of a write. When the read response along with the stored coherence event is received by the DC, it uses information from the original coherence event to create and issue a response to the upgrade request. (CC-ROM action is *send response headers*). The read transaction is retired when read response information from the read TM is accepted by the DCU.

Lastly the DCU controller invokes the ECI TM whenever it has to issue a forward downgrade request. The ECI TM behaves exactly like the read and write TM and sends a forward downgrade transaction through the outgoing channel. The forward downgrade transaction is retired when a forward downgrade acknowledgment is received from the CPU.

8.5.6 Design of DCU's directory: Tag State Unit (TSU)

In this section, we go into the implementation details of DCU's directory, also called TSU. To begin with, we consider the smallest unit of non-distributed memory on the FPGA which is a BRAM. A BRAM can be configured as a true dual-port memory that can hold up to 36K bits of information.

As seen in Figure 8.4, each set has 16 ways with each way storing 36 bits of information. Thus with a single BRAM we can store 64 sets ($64\ sets \times 16\ ways \times 36\ bits\ per\ way = 36K\ bits$) of the 2^{13} sets in the DC's directory.

We earlier made the design choice of having 16/32/64 DCUs per DCS (design-choice 8.12). Thus in a DC with two DCSs, there should be 32/64/128 DCUs. When the 2^{13} sets of the DC's directory (Figure 8.4) is striped across 32/64/128 DCUs, each DCU would have to hold 64/128/256 sets. Since each set has 16 ways and each way stores 36 bits of tag and state information, the directory in the DCU should be able to hold 36K/72K/144K bits of information. This can be fit into 1/2/4 BRAMs.

Thus within a DCU, these sets are stored in one or more Tag-State-Rams (TSRs). Each TSR contains one BRAM to store tag and state information. Depending on the number of sets per DCU we would have 1/2/4 TSRs in a DCU for its directory. Within a TSR, the tag and state information of the 64 sets have to be carefully organized so as to be able to match tags across 16 ways in a given set with lowest possible latency.

Design Choice 8.14. *The directory in the DCU is built upon the smallest unit of non-distributed memory available on the FPGA called the BRAM. Each DCU has one or more BRAMs to store tag and state information.*

8.5.6.1 Tag state ram (TSR)

Each DCU has a number of tag state rams (TSR) to store tag and state information. Each TSR can store data for up to 64 sets, each set having 16 ways. Each way allows to store 18 bits of tag or state information. The DCU has the option of instantiating 1, 2, 4 TSRs thereby allowing 64/128/256 sets per DCU. Depending on this configuration, the ways of a set are split across multiple TSRs. For example, for 64 sets-per-DCU with 1 TSR, all 16 ways of a set are stored in the same TSR. By contrast, for 128 sets-per-DCU with 2 TSRs, ways 0 to 7 of a set are stored in 1 TSR and ways 8 to 15 are stored in the other.

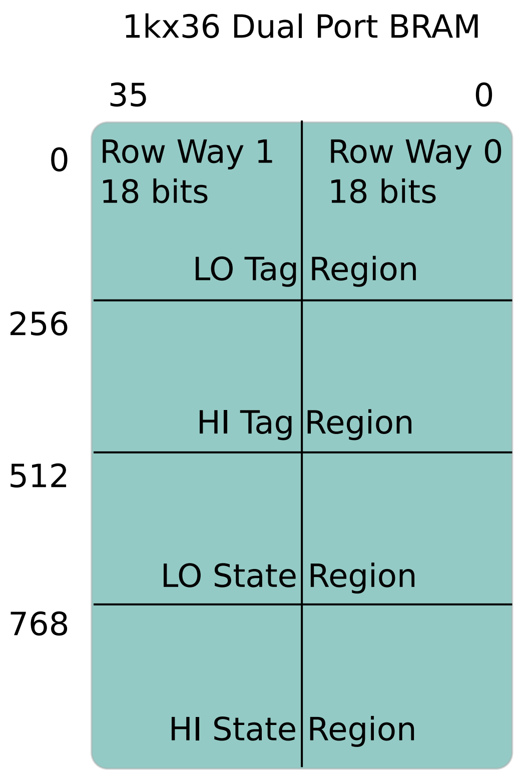


Figure 8.15: *Tag State Ram (TSR): 1 dual port BRAM with 1K rows and each row being 36 bits wide. Two 18-bit ways can be stored per row. The lower half of the BRAM rows store tag information and upper half store the state information. The tag and state regions are further split into LO and HI regions for the two BRAM ports to simultaneously be able to access.*

Each TSR has one true dual port block RAM (BRAM) and is shown in Figure 8.15. The BRAM is configured to have 1024 rows, with each row being 36 bits wide. A single row in the BRAM can hold two 18-bit ways. The 1024 rows are split into two regions: with lower 512 rows forming the tag region to store tags and higher 512 rows store the state, forming state region. A true dual port BRAM has 2 ports for simultaneous reads and writes. To take advantage of this, each tag/state region is further split into LO and HI regions with 255 rows each. Thus the ways of a set are further split across the HI/LO regions. For example in the 64 sets-per-DCU configuration with 16 ways in a TSR, ways 0 to 7 are stored in LO region and 8 to 15 are stored in HI region. For a tag stored in a way in the tag region, its state is also stored in the same way but in the state region. As a result of this organization, only 18-bits of tag information can be stored. This is the reason behind design-choice 8.3, to shrink the 1TiB of FPGA addressable region to 256 GiB.

A TSR allows for 3 types of operation, of which only one operation can occur at a time. The types of operations based on decreasing priority are as follows: write tag and state, read tags, read state.

The write tag and state operation is used to write the tag and state information into a given set and way. The writes are simultaneous with one of the BRAM ports writing to the tag region and the other BRAM port writing to the state region. This operation takes 1 cycle to complete.

The second operation in order or priority is the read tags operation. The read tags operation compares an input tag to all ways in a given set to find a match. An internal state machine iterates through all the ways for a given set in the TSR to find a match for the tag. If there is a hit, the matching way is returned. If there is a miss, an empty way is returned, and if there are no empty ways, a random way is returned. In order to optimize reading, both ports of the BRAM are used to read the tag region, with one port reading the ways in HI TAG region and other port reading the ways in LO TAG region. The state machine terminates early upon hit and worst case performance depends on the number of ways that need to be read. A busy signal indicates when the operation is in progress and goes low upon completion.

The last operation in decreasing order of priority is read state where given a set and way, one of the ports of the BRAM is used to read the state information. This operation takes 1 cycle to complete.

8.5.6.2 Why non-existent memory

Upon a miss, an empty way has to be returned by the TSR so that a new entry can be added if necessary. In order to detect a miss, the tags in all ways has to be read to ensure there is no match and in addition the states would have to be read to make sure that a way is indeed empty. This can be a performance issue, for example, in the case of 64-sets-per-DCU, it would take 4 cycles to ensure there is a miss and a maximum of 4 additional cycles to find a way that is empty. The performance hit due to the first 4 cycles cannot be avoided but the next 4 cycles can be avoided if we can identify that a way is empty just by the contents of the tag. In the current implementation, a tag which is all 1s serve this purpose. We can identify that a way is empty if the contents of the tag in that way is all 1s. This comes at the cost of losing 1 MiB worth of address space. If the 18-bit tag is all 1s, the 13 bit set-index and 7-bit byte offsets are don't cares, thus we lose 20 bits worth of address space for every 256 GiB. If a request is made to the address space where are the tags are 1s then a “non existent memory” (nxm) signal is set.

8.5.6.3 Building TSU with TSRs

The tag state unit (TSU) is the directory that instantiates one or more TSRs for the DCU. It also provides a simpler, standardized interface to interact with the TSRs irrespective of the number of TSRs and how sets and ways between them are distributed.

The TSU provides two operations, one of which can be active at a time. Based on decreasing priority, the two operations are write tag and state for a given set and way, and comparing a given tag to all ways in a set to identify if there is a match and returns its state.

The first operation takes input tag and state information along with set and way information to write the tag and state into its appropriate location in the TSRs. This operation takes 1 cycle and is indicated by a “busy” signal.

The second operation compares an input tag across all ways for a given set. Upon hit, the TSRs are automatically read for the state information and the hit way, state are returned. Upon miss, the state is not read and an empty way is returned. If there are no empty ways available, this is also indicated with a “Set full” signal. In addition, comparing an out of bound tag would be indicated by an nxm signal. This operation takes multiple cycles during which a “busy” signal would be set. It would be cleared upon completion of this operation. It should not be assumed that the state returned upon miss would be the invalid state.

It is to be noted that the TSU does not provide any mechanism to free up ways when a set is full. This is done deliberately because in the current design, we rely on the CPU to perform directory maintenance thereby eliminating the need for the DCU to maintain its directory. This feature is very useful for inducing evictions and recovering directory resources and should be implemented in the future. A previous version of directory had this feature and is discussed in subsection B.2.2.

Takeaway 8.2. *When improving this design of the directory, add mechanism that allows the DC to perform directory resource maintenance.*

8.5.6.4 Optional registering outputs of TSR

As number of TSRs per TSU increases, meeting timing on comparing the tags and aggregating results becomes difficult. To overcome this, there is an option to register the output of each TSR before the results are aggregated in the TSU. This improves timing but at the cost of added latency. The additional latency depends on number of comparisons to be done before a match is found. This stage is optional because for 1 TSR per TSU scenario, it does not affect timing but decreases performance. So care must be taken when using this parameter.

8.6 Customizing the Protocol State Machine

We had seen in previous chapters how a state space exploration tool can take in a specification and automatically generate the DC protocol state machine with all its intermediate states. We had also seen that modifying the protocol can be as simple as changing the specification of ECI and local coherence transactions and using the state space exploration tool to generate a new variant of the DC protocol.

Incorporating this new protocol state machine can be as simple as generating a new protocol CC-ROM and instantiating it in the DCU. To facilitate this, tools have been developed that takes the protocol state machine generated by the state space exploration tool in the from of a CSV file and generate a system-verilog CC-ROM automatically.

8.7 Summary

Thus in this chapter, we looked at the design choices and implementation details that went into designing a distributed DC. The DC is the implementation of the DC protocol state machine and is part of the protocol layer. It should provide the guarantees that are required of the protocol layer and discussed in section 3.4.

The DC is guaranteed to be faithful to the prescriptions of the protocol state machine and thus reproduces its guarantees such as maintaining coherence invariants. The DC provides a separate instance of the protocol state machine for each cache line thereby guaranteeing the coherence transactions on different cache lines are mutually independent.

We looked at the design choices made to make the DC deadlock free whenever resource contention arises, and we also looked at how performance can be achieved by having multiple DCUs running in parallel. We also looked at how the DC's protocol state machine and its interface can be modified to suit the needs of applications. Finally, the DC provides a simplified interface that is used for building the application layer.

One insight that this implementation provides is that even if the CPU's native protocol has never been designed to communicate with anything other than another CPU, we can have a stable and high performance implementation of the DC on an FPGA that runs at a much lower frequency (322 MHz compared to CPU's 2GHz).

9 Evaluation & Applications

9.1 Introduction

Traditional CPU-FPGA acceleration models use the FPGA as an offload accelerator: The CPU provides source and destination descriptors to the FPGA and the FPGA DMA's the source data, processes it, writes results to destination and indicates completion after which the CPU continues operation. This model is followed in heterogeneous systems that do not have a coherent interconnect (e.g. PCIe) between the CPU and FPGA. The drawbacks of this system is as follows. First, the CPU and FPGA cannot operate in parallel on a shared memory region. Second, the source and destination pages would have to be pinned and marked as non-cacheable. Third, by fixing the acceleration model, the interaction between the CPU and FPGA is fixed which undermines the highly re-configurable nature of the FPGA.

As heterogeneous platforms with coherent interconnects between CPU and FPGA become available, the question arises as to what the interface between the FPGA application and the coherent interconnect be? Should FPGA applications also be limited to performing only load and store operations through a cache? Is the only difference between coherent and non-coherent platforms that coherent platforms can provide "fine-grained acceleration". What would a non-traditional acceleration model with coherent interconnects look like?

CCKit provides a clean and high-performance abstraction of a complex coherence protocol to hybrid CPU/FPGA applications that is not based on bulk DMA or non-cacheable register access. This allows us to explore non-traditional acceleration models where user-logic on the FPGA can extend the notion of coherence for applications running on the CPU through components of CCKit such as DC. Applications on the FPGA are not limited to performing loads and stores but are exposed to a much richer interface. For example, through DC's local interfaces, applications can associate multiple unrelated cache lines and address spaces and through DC's memory interfaces, an application can be notified whenever the CPU upgrades or downgrades a cache line. As a result, applications on the FPGA can do much more than just "fine-grained" acceleration.

In section 7.8 we saw how to define an application layer protocol and considerations to avoid starvation and deadlock at the protocol level. In this chapter, we will look at how to implement the application layer protocol on top of the DC. We begin in section 9.2 by describing the interface between FPGA application, DC and byte-addressable memory. section 9.4 talks about the distinction between aliased and un-aliased addressing. We then evaluating the standalone performance of DC's memory and local interfaces through different experimental setups and benchmark applications in sections 9.5.2 through 9.5.4. This gives us an idea of the DC's expected performance under different scenarios. Next, we show two applications that highlight how user-logic on the FPGA can take advantage of access to the coherence protocol. In the first application (subsection 9.6.2) we demonstrate concurrent access of shared memory on the FPGA by threads on both CPU and FPGA. In the second application (subsection 9.6.3) we show how the application can provide a consistent view of two different address spaces. In this application, the notion of coherence is extended from between copies of a cache line to between different cache lines in a manner that is *completely transparent* to CPU software. Both these applications are examples of non-traditional acceleration model where no DMA operations are involved and pages can be cached on the CPU.

9.2 Interfacing FPGA Application to DC

In section 8.2 we saw that the DC exposes a local interface for FPGA applications to interact with the coherence protocol and an AXI interface to access the FPGA address space. In section 7.6 and subsection 7.7.4 we saw the guarantees provided by two variants of the DC protocol (non-locking and locking variants) for an FPGA application that observes

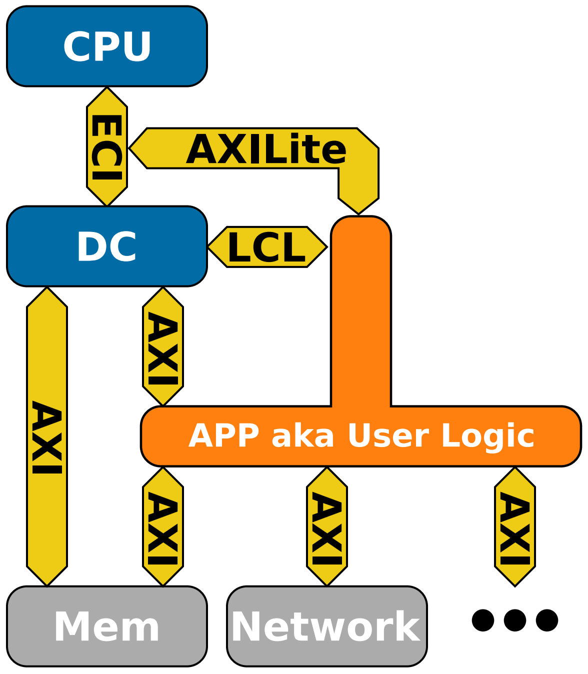


Figure 9.1: *Different configurations in which FPGA applications can be connected to the DC: The application can interact with the DC through its local interface to influence the DC coherence protocol. The application can also tap into the memory channels from the DC if required and extend the view of FPGA address space as seen by software on the CPU. Finally, there is the AXI-Lite configuration interface for IO address space exposed by ECI.*

and issues coherence events through the DC's local and AXI interfaces and section 7.8 discusses the considerations to be taken into account when developing the application layer on top of the DC protocol layer. Applications, depending on the sort of guarantees required, can be connected to either or both these interfaces as shown in Figure 9.1.

Moreover, the DC exposes 256 GiB of FPGA address space through its AXI channels and can be split into several regions as per the needs of an application. For example, the FPGA address space is split into "source" and "view" address regions in case of the materialized-views application described later in subsection 9.6.3. There is also no strict requirement that a byte-addressable memory has to be connected to the AXI channels and, for example, a networking module could be connected as well.

Through such an interface, FPGA applications can serve as memory controllers or accelerators of application-specific coherence and consistency requirements. In contrast, an application that is connected to the coherence protocol through a cache will be able to perform only load and store operations on cache lines.

AXI interface: As seen in design-choice 8.8, the DC can issue a maximum of 32/64/128 outstanding AXI read and write transactions depending on its configuration. But since each DCU is blocking (design-choice 8.12), the actual number of outstanding memory transaction depends on the memory access pattern. The AXI transaction ID of AXI responses should match the ID of the request and can be out-of-order. The DC always reads a full cache line but can write at the granularity of sub-cache-lines (design-choice 8.6). The performance of the DC at this interface is benchmarked in subsection 9.5.3.

Local interface: Although the local transactions have a 6-bit transaction ID (as see in message format in subsection 7.11.1), the actual number of local transactions that can be handled by the DC depends on the access pattern. The response to local requests will have the same ID as the requests and the responses will be out-of-order. There are no ordering guarantees even if same transaction ID is used for different local transactions. The unlock transaction can have its own transaction ID. The performance of the DC at the local interfaces are benchmarked for different access patterns in subsection 9.5.4.

AXI-Lite Interface: As seen in subsection 3.3.1, ECI exposes an AXI-Lite configuration interface that can be used to exchange small amounts of configuration information through the IO address space. The application can take advantage if this bus if necessary.

9.3 Advantages of symmetric protocols over asymmetric protocols

Continuing the discussion in subsection 2.5.1, to look at the advantages offered for application acceleration by symmetric coherent platforms like Enzian over asymmetric platforms, we just need to look at the interface provided for applications on both these platforms. We can see in Figure 2.11 that symmetric platforms allow FPGA applications to interact with the coherence protocol through only loads and stores through a cache. Whereas applications on symmetric platforms have more control and observability over the coherence protocol as shown in Figure 9.1. Special modes like the *bypass* mode in asymmetric platforms is not required in symmetric platforms.

9.4 DC Cache Line Addressing

A 38-bit byte address exposed by the DC to physically address 256 GiB of FPGA address space. This 38-bit address can be split into 31-bit cache line index and 7-bit byte-offset.

Although software on the CPU works with virtual addressing, the address sent over ECI when the CPU accesses the FPGA address space is an *aliased physical address*. The CPU, in addition to performing virtual to physical address translation, also aliases the addresses so as to distribute sequential addresses across parallel tag and data units in its LLC. This is done to optimize sequential access performance. It is to be noted that aliasing a byte address affects only the cache line index and not the byte-offset.

The DC operates in the domain of aliased physical addresses. That is, the cache lines addresses in all coherence messages (both ECI and local) and memory events are aliased physical addresses. FPGA applications can choose to operate in the domain of *unaliased* physical addresses and functions to convert aliased to unaliased and vice versa are available in the ThunderX-1 manual. But, when communicating with the DC, applications should always use aliased physical addresses.

The DC has two DCSs, one for odd (aliased physical) cache lines indices and other for even cache line indices. In ECI, odd VCs carry coherence messages for even cache lines indices and vice versa. The same rule is applied for local VCs as well. Thus the DCS connected to odd VCs will handle even cache line indices and conversely DCS connected to even VCs will handle odd cache line indices. Applications must be careful to issue coherence event for a cache line to the correct VC (odd cache line index goes to even VC number).

Even if the CPU performs sequential reads, the addresses observed on the AXI memory interfaces of the DC will not be sequential (even after unaliasing). This is due to the fact that neither ECI norDC provide any ordering guarantees.

Takeaway 9.1. *The DC always works with aliased physical addresses. Applications can work with unaliased addresses but should convert them to aliased addresses when interacting with the DC. Applications also should be careful in identifying the correct VC to issue a coherence message based on if the cache line is odd or even. The addresses observed on the AXI memory interface of the DC might not be in the same order as it was issued by the CPU.*

9.5 Performance Evaluation of CCKit's DC

To evaluate the performance of DC at its interfaces, we ran a number of micro-benchmarks. Section 9.5.1 describes the implementation details of CCKit and subsection 9.5.2 discusses the experimental setup on which the micro-benchmarks were run and also the resource consumption of the DC on the FPGA. Next we have the first micro-benchmark in subsection 9.5.3 which measures the read, write throughput and latency of the DC at its AXI interface. Finally, we have the second micro-benchmark in subsection 9.5.4 where we measure the throughput and latency of the DC at its local interface. Through these experiments we aim to show that the DC provides a high performance interface for FPGA applications to interact with the coherence protocol.

9.5.1 Implementation details

We have implemented the first version of CCKit on the publicly available Enzian computer [CRS+22, The23], 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, and the other contains a Xilinx VU9P UltraScale+ FPGA [Xil21].

Both CPU and FPGA have 4 channels of DDR4 memory, and the CPU has a 2-level cache with a 16 MiB shared LLC, using 128 B lines. The CPU's native interconnect is exposed to user FPGA logic as the ECI; the 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 230ns (see subsection 9.5.3). Two DDR4 channels (on either node) are sufficient to saturate this link.

On the Enzian machines made available to us, the CPU has 4 x 32 GiB 2133MT/s ECC DIMMs, and the FPGA has 4 x 16GiB 2400MT/s DIMMs. The only parts of Enzian relevant to CCKit are the CPU and FPGA, the ECI link, and the DRAM controllers. The FPGA "shell" provided with Enzian exposes raw inter-socket protocol ECI messages to the FPGA user logic, which lets us build our own coherence state machines above it.

9.5.2 Experimental setup

Figure 9.2 presents the three topologies we use to evaluate CCKit, and compare it to the CPU's native implementation of coherence. As a baseline (*LLC+DDR*) we take a 2-socket

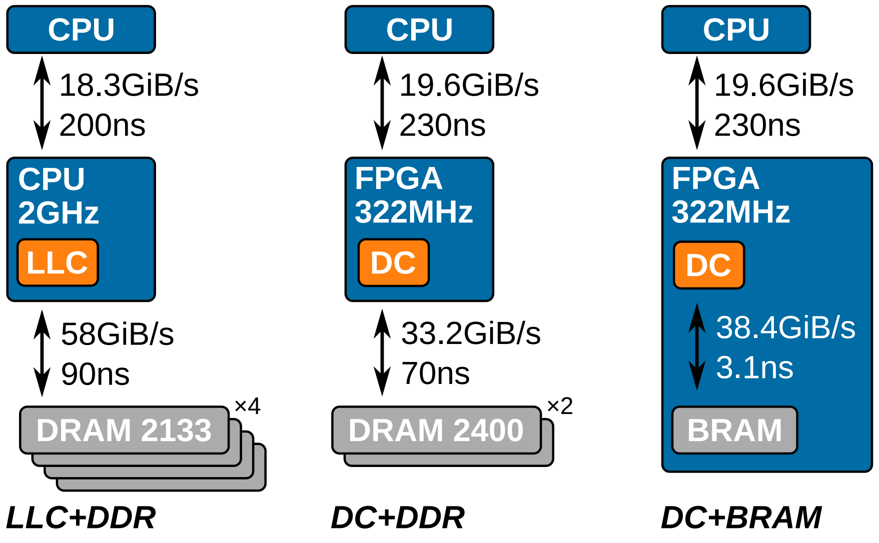


Figure 9.2: *DC configurations and the 2-CPU server: The LLC+DDR configuration is 2-CPU server configuration which serves as the baseline. The DC+BRAM configuration connects BRAMs to the DC's AXI interface and isolates performance of DC. The DC+DDR configuration connects two DDRs to the DC's AXI interface.*

Cavium ThunderX-1 CN8890 server (Gigabyte R150-T61 [GIG23]). Here both sockets are connected by the Cavium Coherent Processor Interconnect (CCPI), the vendor's native implementation. Each CPU runs at 2.0 GHz and each node has four 64 GiB DIMMS (2133 MT/s)

We add two heterogeneous configurations (*DC+DDR* and *DC+BRAM*) in which the second ThunderX-1 CPU is replaced with an FPGA, with ECI replacing CCPI as the interconnect, using the CCKit DC described in chapter 8. In the *DC+DDR* configuration, the FPGA connects the AXI interface of each of the two DCSs to one 16 GiB DIMM (2400 MT/s) with a standard Xilinx DRAM controller IP [Xil22a]. *DC+BRAM* replaces the DRAM with two 64 KiB BRAMs, to isolate the performance of the DC from that of the DRAM IP.

The round-trip latency and throughput figures in Figure 9.2 are the measured performance of the existing hardware (for *LLC+DDR*), or the ECI implementation supplied with Enzian (for *DC+DDR* 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 the CCKit DC as fixed parameters of the underlying platform. ECI shows 7% higher throughput at 15% higher latency than the native ThunderX-1 implementation.

Configuration	LUT (%)	CLB (%)	BRAM Tile (%)
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

Table 9.1: *Resource Utilization Footprint on FPGA.*

In all configurations, the DC is configured to have 2 DCSs with 64 DCUs each. Table 9.1 shows resources utilized by different components and configurations of CCKit, all 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 "MIGs"). Even with 32 DCUs per DCS, CCKit leaves 70% of the FPGA resources for applications.

9.5.3 DC read-write throughput and latency

While the supplied ECI implementation is comparable to the CPU's own, the DC adds symmetric coherence between the two sockets. In this section, we evaluate what, if any, overhead the DC itself introduces. For all three configurations in Figure 9.2, 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. All throughput tests use two CPU cores.

Figure 9.3 presents throughput for all combinations. Each bar is the mean of 100 runs, with 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 using BRAM is replaced with DRAM. The cause appears to be the known inefficiency of the Xilinx MIG IP non-sequential access patterns [Xil22a]. This results from the ThunderX-1 LLC address scrambler described in chapter 8 transforming sequential reads to a pseudo-random pattern. Fully random reads further stress the MIG's transaction scheduler and begin to cause contention on DCUs, leading to a moderate slowdown relative to sequential.

Sequential and random writes perform similarly, and are broadly consistent with random

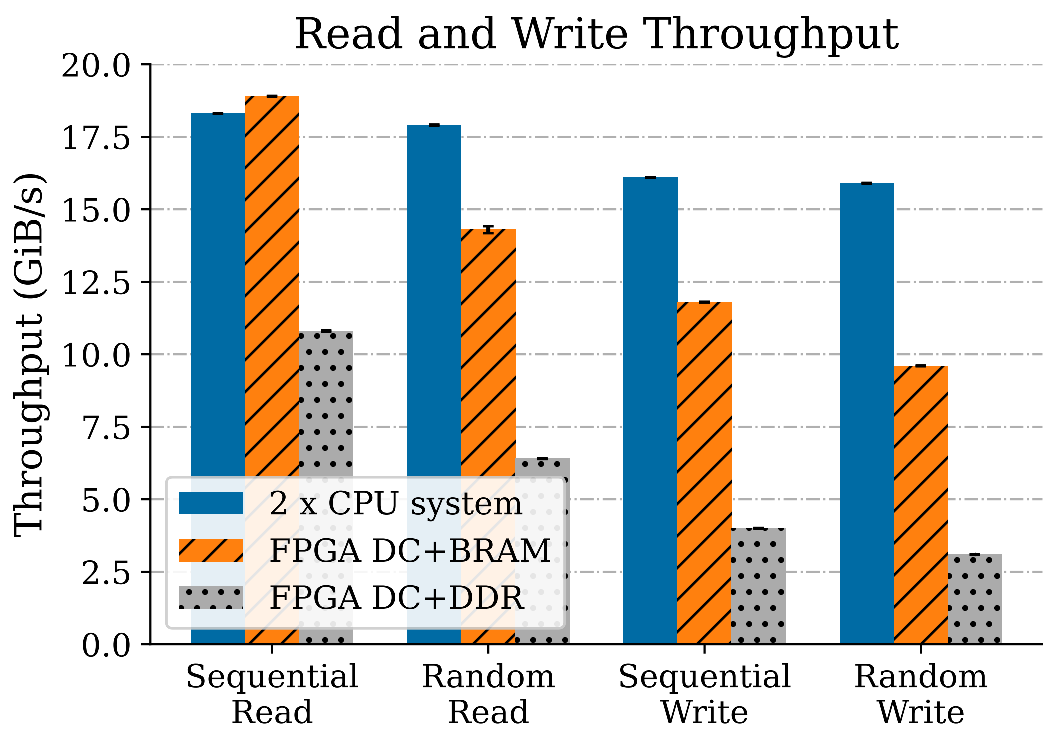


Figure 9.3: *CCKit DC performance vs. two-CPU system.*

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

Table 9.2: *Sequential and Random Read Latency*

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.

Table 9.2 shows the average round-trip latency of reads for all three configurations. These are the average of 15 runs over the full 1GiB, with one access per cache line and no prefetching to ensure serialization. Comparing *LLC+DDR* with *DC+DDR* indicates that CCKit adds 323–330ns (120–122%) 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 173–186ns, with the remaining 137-157ns due to the DRAM latency and the AXI interconnect. A write instruction consists of a read to the LLC followed by a CPU write which commits as soon as it hits the write buffer, and as such has the same latency characteristics as a read.

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

9.5.4 DC clean-invalidate throughput and latency

In addition to allowing the CPU to coherently cache FPGA-homed data, the DC provides a simplified request-acknowledge interface so that applications on the FPGA 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 paper rely on these operations and in this section we evaluate their performance.

For both *DC+BRAM* and *DC+DDR* configurations, the CPU first reads 8MiB of sequential data into its LLC in shared state (clean), 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 will be higher than indicated by per-request latency as it benefits from pipelining.

The integration of FPGA application with the DC for this experiment is shown in Figure 9.4. The application monitors the AXI interface to the memory to check if the CPU has upgraded 8MiB worth of cache lines. Once the count is reached, the application starts clean-invalidating cache lines through the DC's local interface. The application has a counter to hold the next unaliased cache line index to be invalidated. This cache line index is aliased and padded with 7 zeros for byte offset to generate the address that is sent to the DC as part of clean-invalidate request. The non-locking variant of DC was used since the application did not require any locking capabilities.

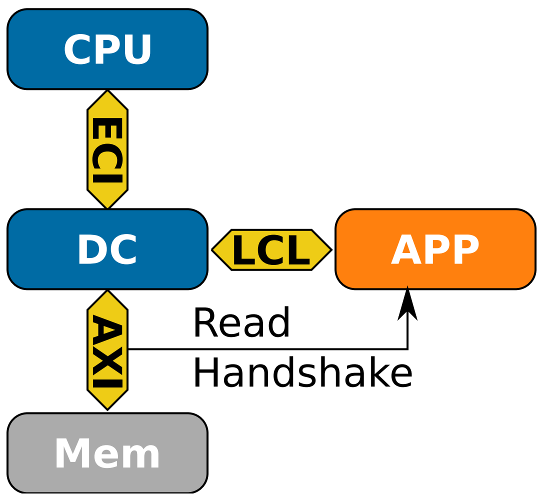


Figure 9.4: *Architecture to measure DC clean-invalidate performance*

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

Table 9.3: *Directory Controller Clean-Invalidate Performance*

Throughput and latency are shown in Table 9.3. The latency of a single invalidate is the same in both cases, at 350ns, but throughput varies. The throughput is bounded above by the rate at which the application can issue requests (1 per clock at 322MHz), and below by the time needed to write back 8MiB of dirty 128B cache lines at the measured ECI throughput of 20GiB/s: around 150×10^6. 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 DCU.

9.6 Applications

Applications that take advantage of CCKit have a software component that runs on the CPU and a hardware component that runs on the FPGA. Specifically with the DC, both these components operate on FPGA-homed cache lines with the CPU-side interface discussed in subsection 1.2.5 and the FPGA component interfacing with the DC.

In this section we first look at the steps to consider when developing applications using DC. Then we look at two example applications.

9.6.1 Steps to consider when developing applications

The following steps might be useful to consider by an application developer when developing with the DC. These are just guidelines, what is useless for the carpenter might be useful for the tree.

1. **Identify components and interaction between them:** To begin with, the developer has to segment the application into a software part that runs on the CPU and the user logic that runs on the FPGA. In this stage, the developer also considers how the software component on the CPU and user logic on the FPGA would interact with each other. The interaction can happen either through coherence messages, or through the slower AXI-Lite configuration interface, or both.

   The software on the CPU interacts with the CPU's LLC through loads, stores and invalidations. This interaction might generate coherence messages that would be handled by the DC and observed by the user logic.

   At this stage the application developer can check if the interface provided by the DC is sufficient to provide the observability required by user logic on the FPGA and if not how can the interface and the protocol specification be enhanced to do this.

2. **Identify application specific invariants:** The next step is to identify the invariants that need to be guaranteed by the user logic on the FPGA to software on the CPU for this application. These invariants can be in the form of correlating states of cache lines or address spaces so as to transparently provide certain guarantees to the software component.

3. **Check guarantees required at DC interfaces:** Next the developer can identify the guarantees that are needed with respect to states of cache lines to satisfy the chosen invariant and to check if these guarantees can be achieved by interacting with the DC.

   We have seen in section 7.6 and subsection 7.7.4 how modifying DC protocol can provide different guarantees on states of cache lines. The application developer can decide if the guarantees provided by default protocol variants (Figure 7.1 or Figure 7.2) are sufficient or if changes would have to be made.

   The developer also can consider how the user logic would interface with the DC and memory or any other module.

4. **Develop protocol state machine:** The developer can then develop the protocol with which the application would interact with the DC to satisfy the invariants. At this point the developer is not concerned about how to actually implement the protocol but rather the protocol state machine (check subsection 3.4.1 to know the difference between a protocol state machine and its implementation). How such a protocol state machine can be developed is described in section 7.8.

   The requirements of application protocol state machine are as follows. First, the application protocol state machine should guarantee the invariants required by the application. Second, there should not be any protocol deadlocks. In other words the protocol should not wait for a message that it would never receive. Third, it can be optimized for performance but that is up to the developer. If the developer needs high performance both the protocol state machine and its implementation should be performant.

   The developer can also explore if the application protocol can be simplified by enhancing the DC. For example adding new local transactions. To reiterate, any new coherence transaction that does not create dependencies between cache lines (i.e. coherence transaction that involves only one cache line) can be pushed into the DC protocol layer to simplify the application layer protocol.

5. **Implement protocol state machine:** At this stage, the developer decides how the protocol state machine would be implemented on FPGA. This is the user logic that interacts with the DC and should provide all guarantees that is provided by the protocol state machine. In addition, the implementation should also consider deadlocks that arise due to limited resources (resource deadlocks) and performance.

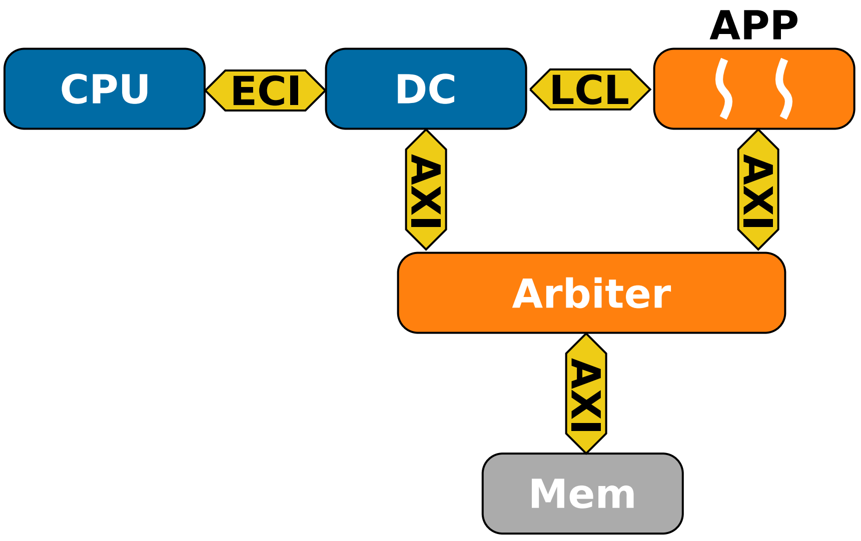


Figure 9.5: *Architecture for concurrent access to shared data*

9.6.2 Concurrent access to shared data structures

In this first use case, we demonstrate how to use CCKit to enable threads on the CPU and the FPGA to concurrently work on a shared data structure while maintaining coherency. As future work, we plan to take this use case as the basis for distributed computing where the data is being 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.

The shared data structure we use in this experiment is a table homed in FPGA memory (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 increments 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 1s 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.

Figure 9.5 shows the integration of DC and FPGA application for this use case. The application has two threads (one for odd cache lines and other for even) with each thread having a single outstanding read-modify-write operation. The DC on the FPGA implements the locking version of directory protocol which allows threads on the FPGA to clean-invalidate and lock cache lines. Once clean-invalidate operation is completed, the state of the cache line in 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. In order to avoid starvation due to locking (subsection 7.7.5), the read-modify-writes are serialized one after another. The application protocol is shown in Figure 9.6.

Figure 9.7 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 FPGA (CPU-FPGA line) or the contention thread (CPU-CPU line) respectively. Throughput is given in millions of rows per second, and we report average and standard deviation for 10 iterations. In the CPU-FPGA configuration the CPU reaches about 45.5 million rows per second without contention. This gradually degrades to about 2.5 million rows per second when the FPGA contends for the entire table. In the CPU-CPU configuration the respective numbers for the access thread are about 41 million rows per second with no contention and 5 million rows per second with maximum contention.

We can see from the shape of the curves in Figure 9.7 that the CPU-FPGA setup using CCKit behaves very similarly to an off-the-shell 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.

9.6.3 Maintenance of materialized database views

In the third use case, we offload view maintenance as used in a relational database to the FPGA and use the 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 common for a range of purposes: access control, simplifying query development, and performance optimization by already pre-computing some parts of common queries. The use case illustrates how the coherence protocol can be tailored to a particular appli-

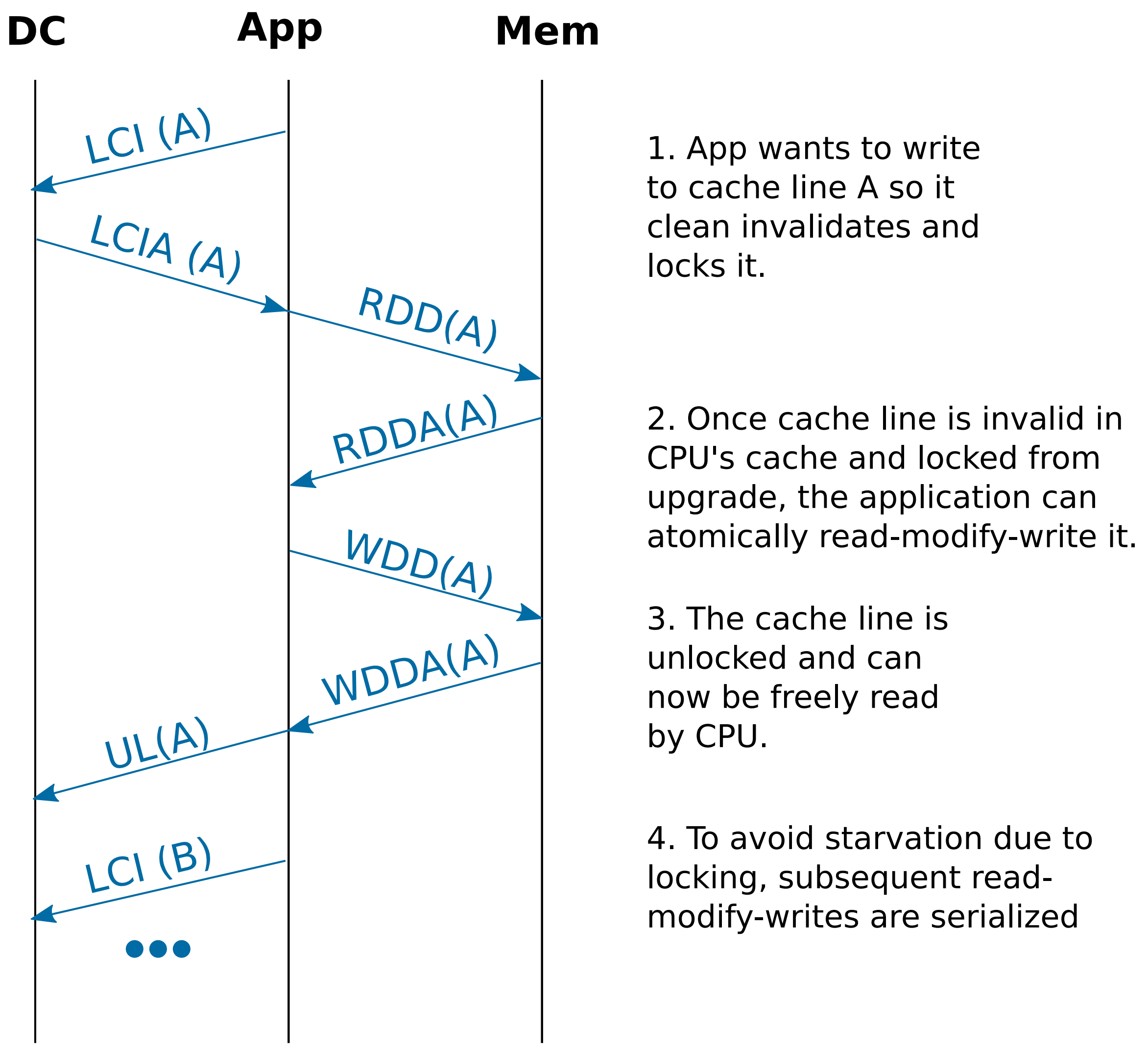

DC
App
Mem
LCI (A)
LCIA (A)
RDD(A)
RDDA(A)
WDD(A)
WDDA(A)
UL(A)
LCI (B)
1. App wants to write to cache line A so it clean invalidates and locks it.
2. Once cache line is invalid in CPU's cache and locked from upgrade, the application can atomically read-modify-write it.
3. The cache line is unlocked and can now be freely read by CPU.
4. To avoid starvation due to locking, subsequent read-modify-writes are serialized

Figure 9.6: *Shared data access: application protocol.*

cation to exploit the FPGA's ability to control the CPU's cache. We use the fact that CCKit provides access to the cache coherency protocol messages to trigger operations on the FPGA that take care of the 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

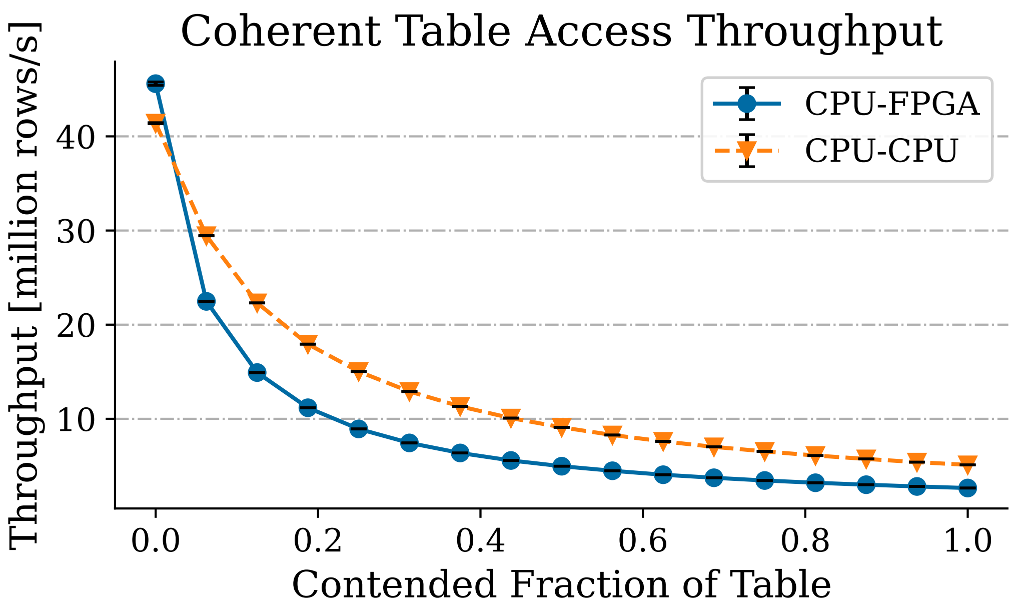


Figure 9.7: *Shared data access: CPU–CPU vs. CPU–FPGA.*

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 and sorts the result by the customer key. The materialized view is stored in a second coherent address range backed by BRAM on the FPGA.

The offloaded view maintenance 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 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 address used for synchronization, triggering an invalidation message to the FPGA that is then used as the signal to run the operator maintaining the view.

This highlights the advantage of a customizable coherence protocol running on the DC. Compared to the operator in subsection 9.6.2, the materialized view operator does not require locking of cache lines that are invalidated. A customized coherence protocol allows

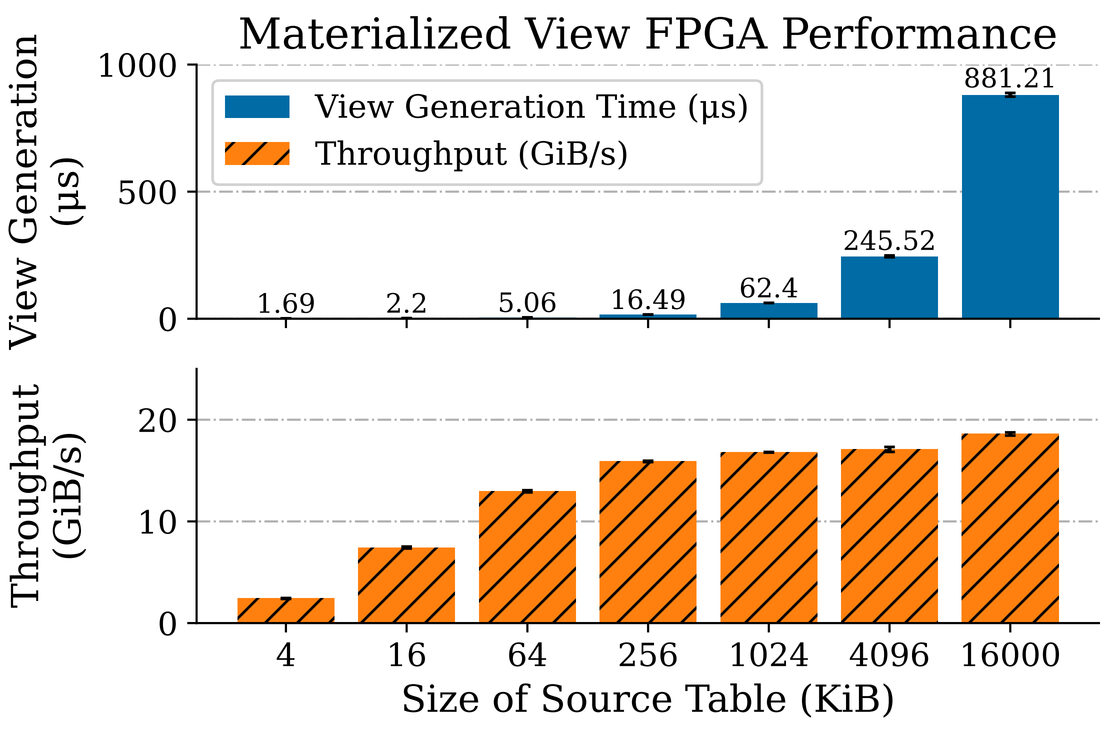


Figure 9.8: *View materialization performance.*

us to switch between protocol versions with and without locking capabilities. Removing locking and the associated state from the protocol simplifies the design of this operator.

Figure 9.8 (*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. We did not do this to simplify the interpretation of the results.

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.

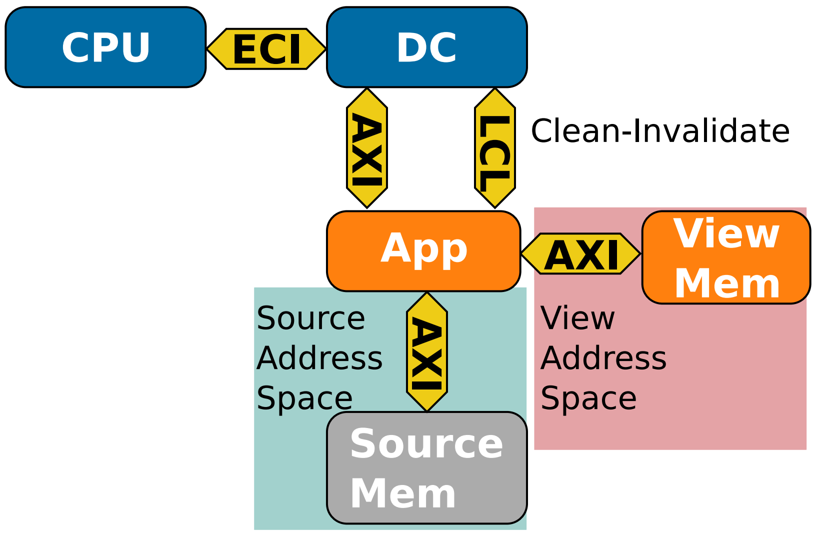


Figure 9.9: *View materialization architecture.*

9.6.4 Implementation details

Algorithm 7 shows the pseudocode of a single CPU thread. The CPU first appends new entries to the source table by reading source cache lines in exclusive and modifying them in its LLC. Once the appends have completed, there is a barrier and the CPU reads a pre-defined synchronization address. The barrier ensures that all appends have completed before the CPU reads the synchronization address. Once the read response arrives, the results of the query have been calculated by the FPGA and are available for the CPU to read. After reading the views, the CPU can go back to appending source tables. It is to be noted that the CPU does not perform any calculations required for the query but the results are available to them transparently.

On the FPGA side, the FPGA application interfaces with both AXI and local channels of the DC as shown in Figure 9.9. Depending on the address observed in the AXI channels, the application splits the FPGA address space into source and view address spaces, with a memory backing each. In our case, an address with most-significant-bit as 0 is source memory and 1 is view memory.

When the CPU starts appending to the source table, it reads an address from the source memory in exclusive. When the application observes the first source read request, it stores the start address and invalidates any copy of the synchronization address residing in CPU's LLC. This invalidation ensures that the FPGA will get a notification (in the form of an

Algorithm 7 Materialized view pseudocode for software on CPU: CPU appends to source table and then reads a synchronization address. When the sync address read returns, the materialized view can be freely read.

```
1: $append\_src$(O_CUSTKEY, O_TOTALPRICE)
2: $append\_src$(O_CUSTKEY, O_TOTALPRICE)
3: $\cdots$
4: $append\_src$(O_CUSTKEY, O_TOTALPRICE)
5: $barrier()$
6: $read\_sync\_addr()$
7: $barrier()$
8: $read\_view$(O_CUSTKEY)
9: $read\_view$(O_CUSTKEY)
10: $\cdots$
11: $read\_view$(O_CUSTKEY)
12: $barrier()$
13: $append\_src$(O_CUSTKEY, O_TOTALPRICE)
14: $\cdots$
```

upgrade request) for any subsequent reads of the synchronization address by the CPU. The FPGA application then provides coherent access to the source memory and keeps track of number of appends performed.

As appends to source table proceed, the source cache lines with dirty data could be voluntarily downgraded by the CPU. The application has to ensure that this dirty data is written back to the source memory and the view memory is updated to reflect the changes made to source memory. Changing the contents of a view cache line in its memory makes any copies of it in the CPU's LLC stale. The application also makes sure that this stale view gets invalidated.

Once appends have completed, the CPU reads the pre-defined synchronization address. This upgrade request is observed by the FPGA application and serves as a notification to make the view table consistent with updates to the source table. The application stalls this read request and starts *cleaning* the source cache lines from the CPU's cache based on previously stored start address and number of appends made.

As source cache lines get cleaned, the application writes back and dirty data and invalidates any stale views as previously described. When acknowledgements for all clean and clean-

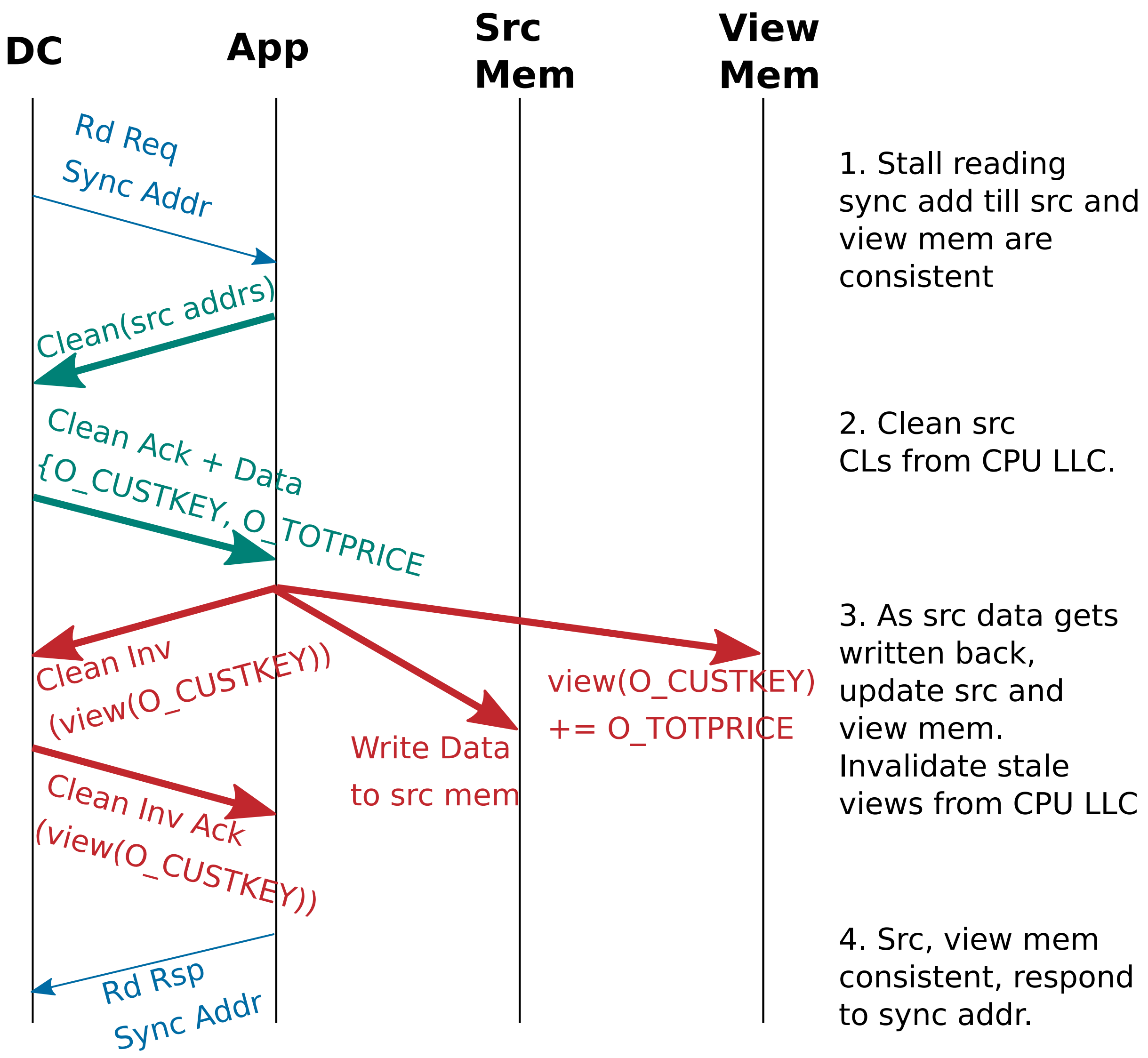

DC
App
Src Mem
View Mem
Rd Req
Sync Addr
Clean(src addrs)
Clean Ack + Data
{O_CUSTKEY, O_TOTPRICE
Clean Inv
(view(O_CUSTKEY))
Write Data
to src mem
view(O_CUSTKEY)
+= O_TOTPRICE
Clean Inv Ack
(view(O_CUSTKEY))
Rd Rsp
Sync Addr
1. Stall reading sync add till src and view mem are consistent
2. Clean src CLs from CPU LLC.
3. As src data gets written back, update src and view mem. Invalidate stale views from CPU LLC
4. Src, view mem consistent, respond to sync addr.

Figure 9.10: *View materialization application protocol.*

invalidate requests arrive from the DC, the application responds to synchronization address indicating that the results of the query are available in the view memory. The protocol of the application is shown in Figure 9.10.

The DC implements the non-locking variant of local clean and clean-invalidate transactions since the application does not require any locking capabilities.

9.7 Summary

In this chapter, we show how the DC provides an abstracted interface for user logic on the FPGA to interact with the coherence protocol. The advantages of such an interface provided by symmetric platforms over the interface provided by asymmetric platforms were discussed. We also benchmarked the DC to show that it provides reasonable performance at its interface for applications. Then we looked at how a developer can approach developing applications along with two sample applications that showcase acceleration models different from the traditional DMA based FPGA acceleration.

10 Conclusions

10.1 Summary

This dissertation provides a layered approach to build a customizable cache coherence stack on the FPGA that aims to provide coherent access for applications on the CPU and FPGA to the FPGA attached memory. The stack is built on Enzian which is a symmetric heterogeneous CPU-FPGA platform where both nodes are connected by a coherent interconnect and each node is responsible for maintaining the coherence of its own memory.

The lower most layer is the ECI layer that allows CPU and FPGA to exchange coherence messages. It is designed to be deadlock free and guarantees delivery of messages but does not guarantee any ordering.

The DC protocol layer is built on top of the guarantees provided by the ECI layer. The DC protocol layer maintains coherence invariants at the granularity of a cache line and guarantees to treat each cache line are mutually independent from other cache lines. It accounts for conflicts that can arise due to ECI latency and reordering of coherence messages by ECI. It is deadlock free, optimized for performance in the critical path and provides an abstracted interface for applications to the coherence protocol.

On top of the DC protocol layer we have the application protocol layer that can extend the notion of coherence across cache lines and even address spaces. Each application can guarantee their own set of invariants, should be deadlock free and performant.

The focus of this dissertation is on DC protocol and application protocol layers. Each protocol layer is split into two components namely the protocol state machine and its implementation. Both components should provide guarantees required for that layer, should account for protocol and resource deadlocks, and be optimized for performance.

For DC protocol layer there is no formal specification describing the protocol. So a protocol model was built by reverse engineering the coherence protocol from traces observed between two CPUs. This model is then used to iteratively identify all coherence transactions that would be part of the DC protocol layer. First, the coherence transactions that would be initiated by the CPU to access FPGA memory were identified. Next, the coherence transactions that can be initiated by the DC to the CPU's LLC were identified and Finally, the coherence transactions that applications can use to interact with the DC were defined.

For all these transactions, a way of specifiying them was developed. With carefully observing these coherence transactions, we were able to identify design choices for building a state space exploration tool that would maintain coherence invariants, be deadlock free and optimized for performance in the critical path. The state space exploration tool takes in a specification and automatically generates a state machine with intermediate states in the form of a state table. This state table can then be implemented on the FPGA to build the DC protocol layer.

We also looked at how to implement the DC protocol state machine on the FPGA, the design choices that were made for performance and also to provide guarantees required by this layer. This implementation was evaluated and found to provide reasonable performance: It can saturate the ECI bandwidth and has a sub 400ns round-trip latency. Moreover, it provides high performance interfaces that allows user logic on the FPGA to influence the coherence protocol. A unique insight here is that we can have a stable and high performance implementation of the DC on a slow running FPGA interacting with the CPU's native coherence protocol, even if the CPU's protocol was never designed to communicate with anything other than another CPU.

A main feature of both the protocol state machine and its implementation is their customizability. The protocol is specified in a machine readable format and can be easily modified. Once the modifications are made, the state space exploration tool, with minimal or no alteration, can generate the state table. Tools are available to convert the state table to the CC-ROM hardware module which can then be plugged into the existing DC implementation without any or minor modifiecations. The advantage of having such customizability is that certain features that are required by the application layer can be pushed into the DC protocol layer easily thereby simplifying the application layer protocol.

The thesis then focuses on the acceleration model that is enabled by such a system. Applications can interact with the coherence protocol to transparently extend the notion of coherence to software on the CPU. This acceleration model is compared to the DMA based acceleration mode on non-coherent platforms as well as the load-store model in asymmetric coherent platforms. One insight is the abstracted interface provided by the DC on Enzian allows a larger class of applications than the other acceleration models. This is demonstrated with application examples.

10.2 Directions for Future Work

10.2.1 Fixing issues with the current work

A few less-than-ideal design choices were made which could be fixed in the future. The known issues are compiled below.

1. The application initiated local transactions could be more formally described (section 7.5 and subsection 7.5.1).
2. Additional local coherence messages are needed to avoid starvation that can be caused by locking cache lines (subsection 7.7.5).
3. Avoiding potential resource deadlock situations in DC's implementation that have not yet manifested (Takeaway 8.1).
4. Adding features to DC's directory that allows the DC to perform directory resource maintenance in addition to relying on the CPU. This might be important especially if local coherence transactions occupy empty slots in the directory (Takeaway 8.2)

10.2.2 Formally specifying and verifying the protocol

An informal approach has been taken throughout the thesis with respect to correctness of the specification, the state machine and its implementation (for example section 5.3). A more formal approach might be interesting and valuable.

10.2.3 Relaxing fundamental design choices

Allowing FPGA user logic to perform coherent caching operations on FPGA memory: The protocol model and specification have their scope tied to fundamental design decisions discussed in subsection 3.2.2 such as not allowing coherenct caching operations by FPGA applications. Removing this restriction can be used to study what other abstractions can be provided by the DC to user logic on the FPGA.

Providing user logic on the FPGA with coherent access to CPU memory (CC): As discussed in subsection 1.2.3 a CC is needed on the FPGA for FPGA applications to coherently access CPU memory. Traditionally CC provides a load-store interface for applications through a cache. It might be interesting to explore the abstractions that can be provided by a CC that doesn't use a cache. For example, the CC in CCKit can provide a coherence transaction that allows FPGA applications to read-modify-write a cache line without a cache. It can also provide a coherence transaction that does read-and-hold where the application would read a CPU-homed cache line in Exclusive and holds it till a downgrade notification comes from the CPU.

10.2.4 Explore applications

The guarantees that are provided by the DC on states of cache lines can be very useful to provide persistence guarantees. Applications can also extend the materialized view application to provide different coherent views of data. For example, coherent row and column views where modifications to the row major view will be transparently visible on the column major view as well. More applications that take advantage of the coherence protocol are described in section 1.3.

A
Miscellaneous Specifications and State Diagrams

This chapter gives the specification digarams for F31 forward downgrade transactions. The DC has issued forward downgrade request to downgrade from E to I and is waiting for a response. It also shows the state diagram for DC protocol state machines with and without locking capabilities along with a one-to-one mapping between the names of coherence messages used in this thesis to the actual coherence message names used by the CPU's native protocol.

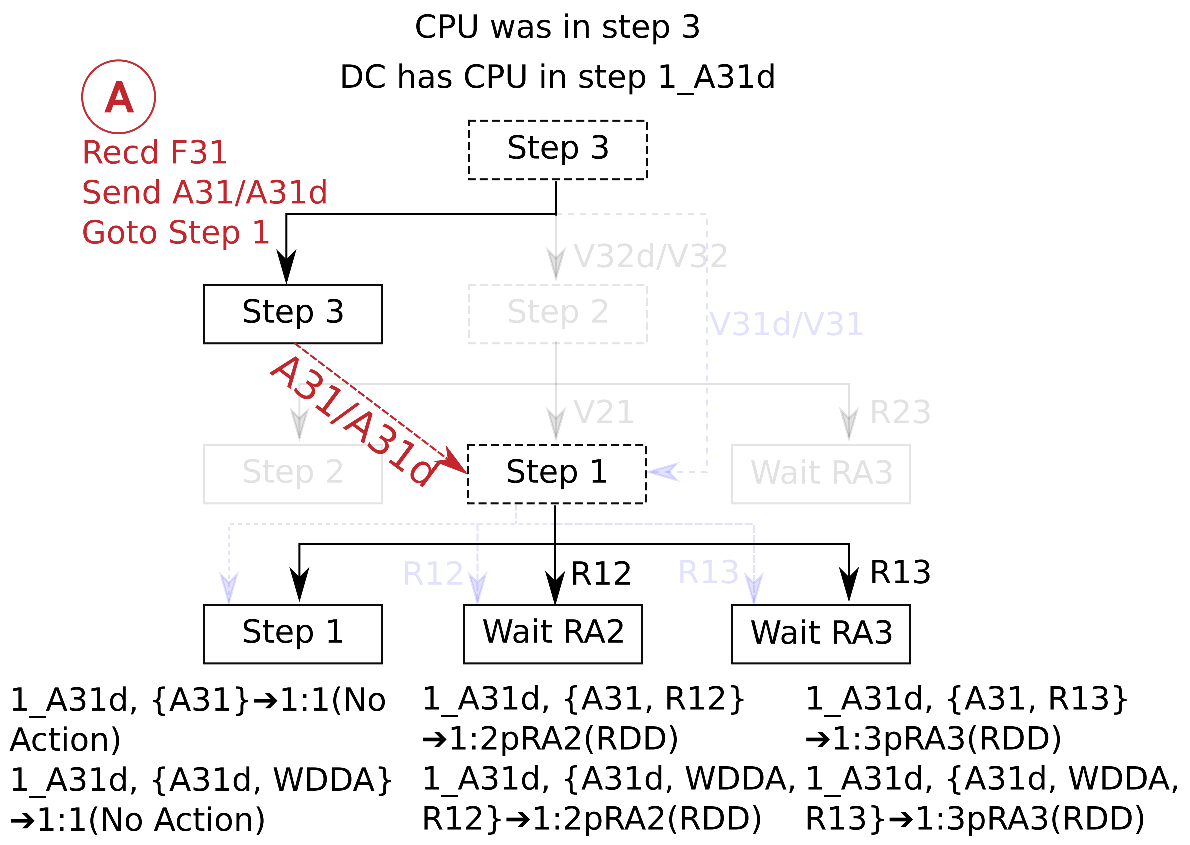

CPU was in step 3
DC has CPU in step 1_A31d
A
Recd F31
Send A31/A31d
Goto Step 1
Step 3
Step 3
V32d/V32
Step 2
V31d/V31
A31/A31d
V21
R23
Step 2
Step 1
Wait RA3
R12
R12
R13
R13
Step 1
Wait RA2
Wait RA3
1_A31d, {A31}➔1:1(No Action)
1_A31d, {A31d, WDDA} ➔1:1(No Action)
1_A31d, {A31, R12} ➔1:2pRA2(RDD)
1_A31d, {A31d, WDDA, R12}➔1:2pRA2(RDD)
1_A31d, {A31, R13} ➔1:3pRA3(RDD)
1_A31d, {A31d, WDDA, R13}➔1:3pRA3(RDD)

Figure A.1: *Pathways CPU can take when F31 is received, scenario A: The CPU receives F31 when it is at step 3. The CPU issues A31 (or A31d if dirty) and comes down to step 1. At step 1, it can continue to remain there or make upgrade requests to step 2 (R12) or step 3(R13). The specification state equations are given below each pathway.*

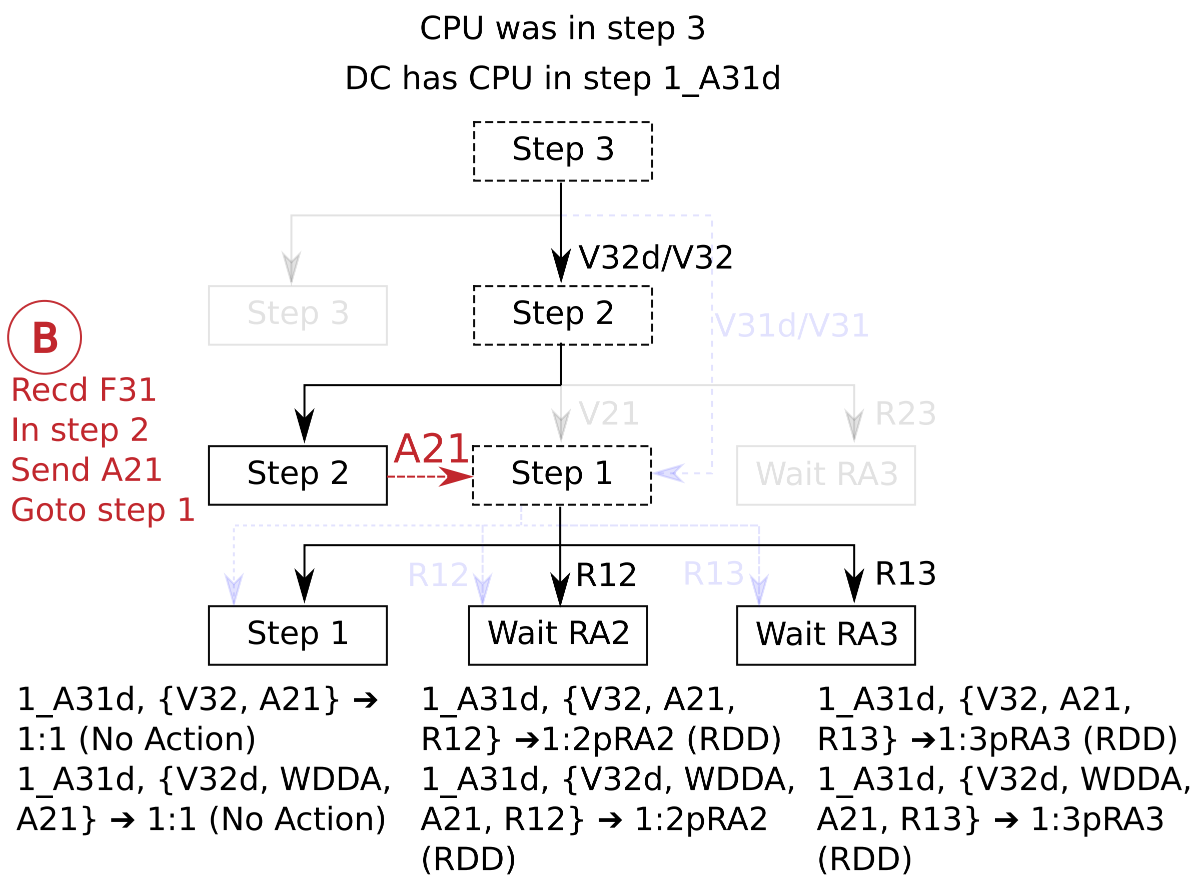

CPU was in step 3
DC has CPU in step 1_A31d
Step 3
V32d/V32
Step 3
Step 2
V31d/V31
B
Recd F31
In step 2
Send A21
Goto step 1
V21
R23
Step 2
A21
Step 1
Wait RA3
R12
R12
R13
R13
Step 1
Wait RA2
Wait RA3
1_A31d, {V32, A21} ➔ 1:1 (No Action)
1_A31d, {V32d, WDDA, A21} ➔ 1:1 (No Action)
1_A31d, {V32, A21, R12} ➔1:2pRA2 (RDD)
1_A31d, {V32d, WDDA, A21, R12} ➔ 1:2pRA2 (RDD)
1_A31d, {V32, A21, R13} ➔1:3pRA3 (RDD)
1_A31d, {V32d, WDDA, A21, R13} ➔ 1:3pRA3 (RDD)

Figure A.2: *Pathways CPU can take when F31 is received, scenario B: The CPU has stepped down from step 3 to 2 (V32d/V32) when it receives F31. Since the CPU is at step 2, it issues A21 and comes down to step 1. At step 1, it can make upgrade requests to step 2 (R12) or step 3 (R13). The specification state equations are shown below each pathway.*

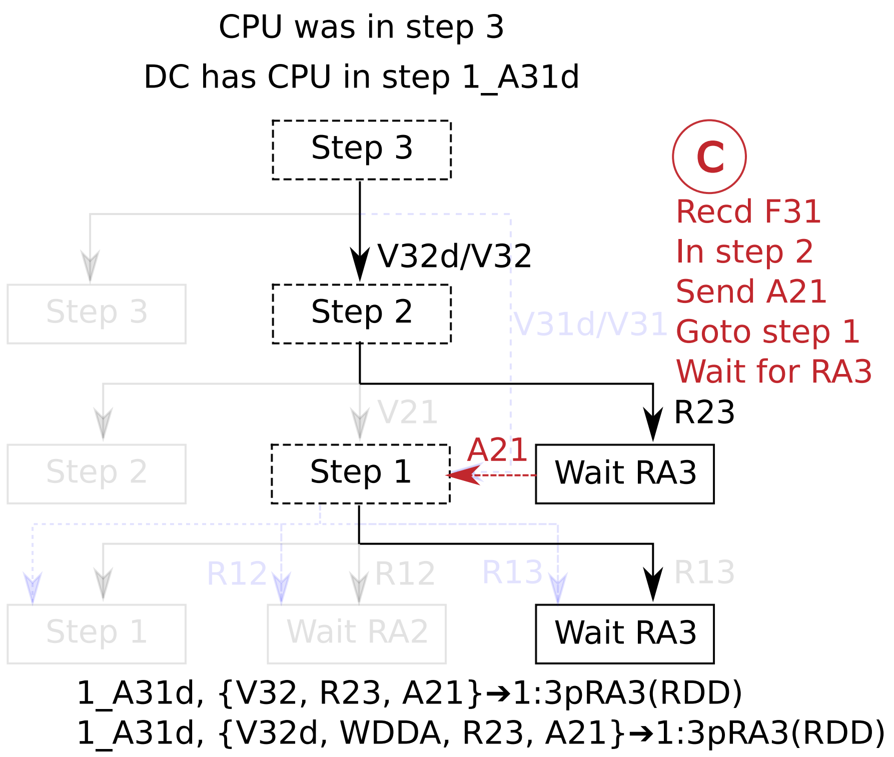


Figure A.3: *Pathways CPU can take when F31 is received, scenario C: The CPU has come down from step 3 to 2 (V32d/V32) and has made an upgrade request R23. It receives F31 when waiting for a response. Since CPU is in step 2, it issues A21 to come down to step 1 and continues to wait for a response to the upgrade request. The specification state equations are shown below each pathway.*

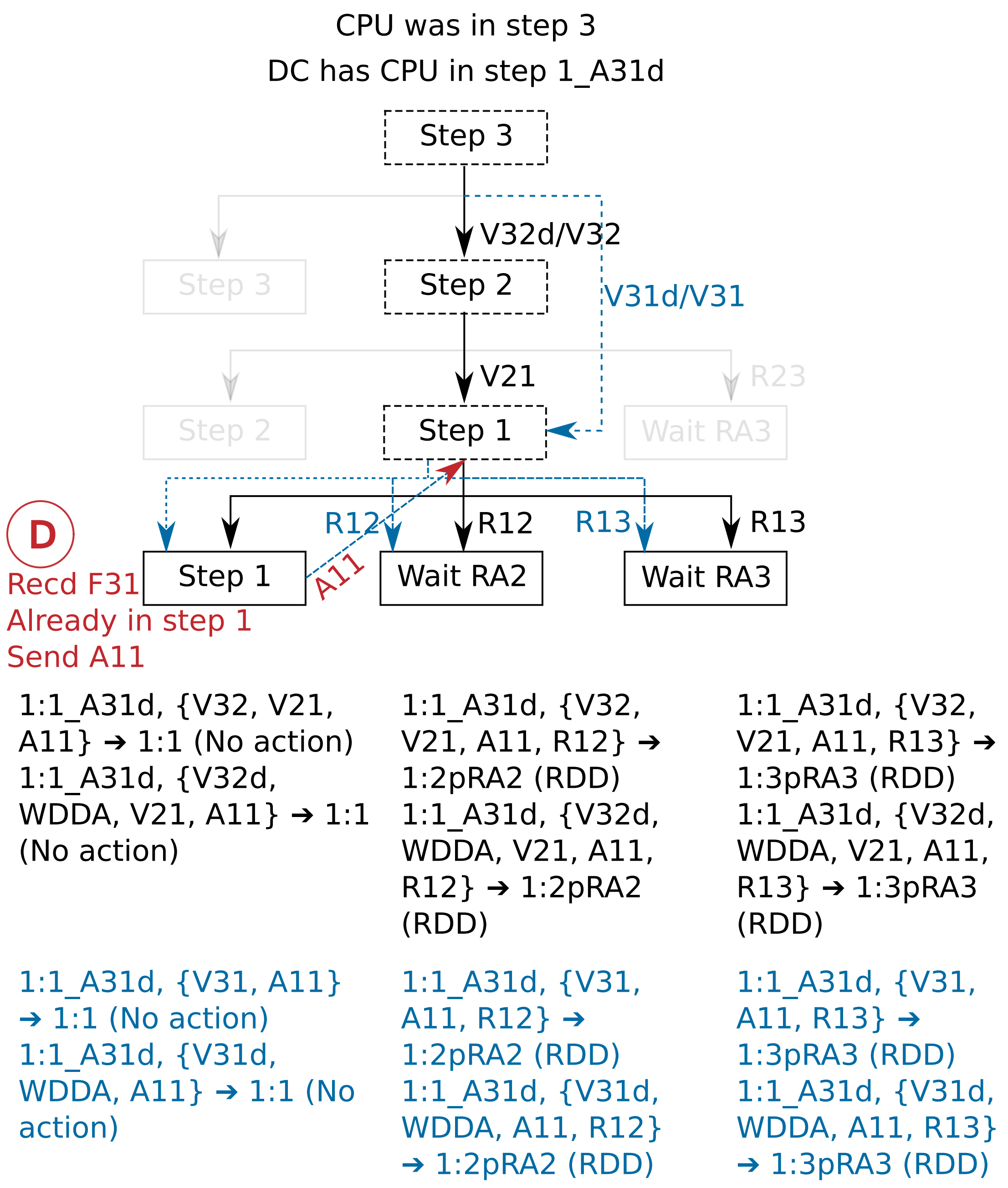
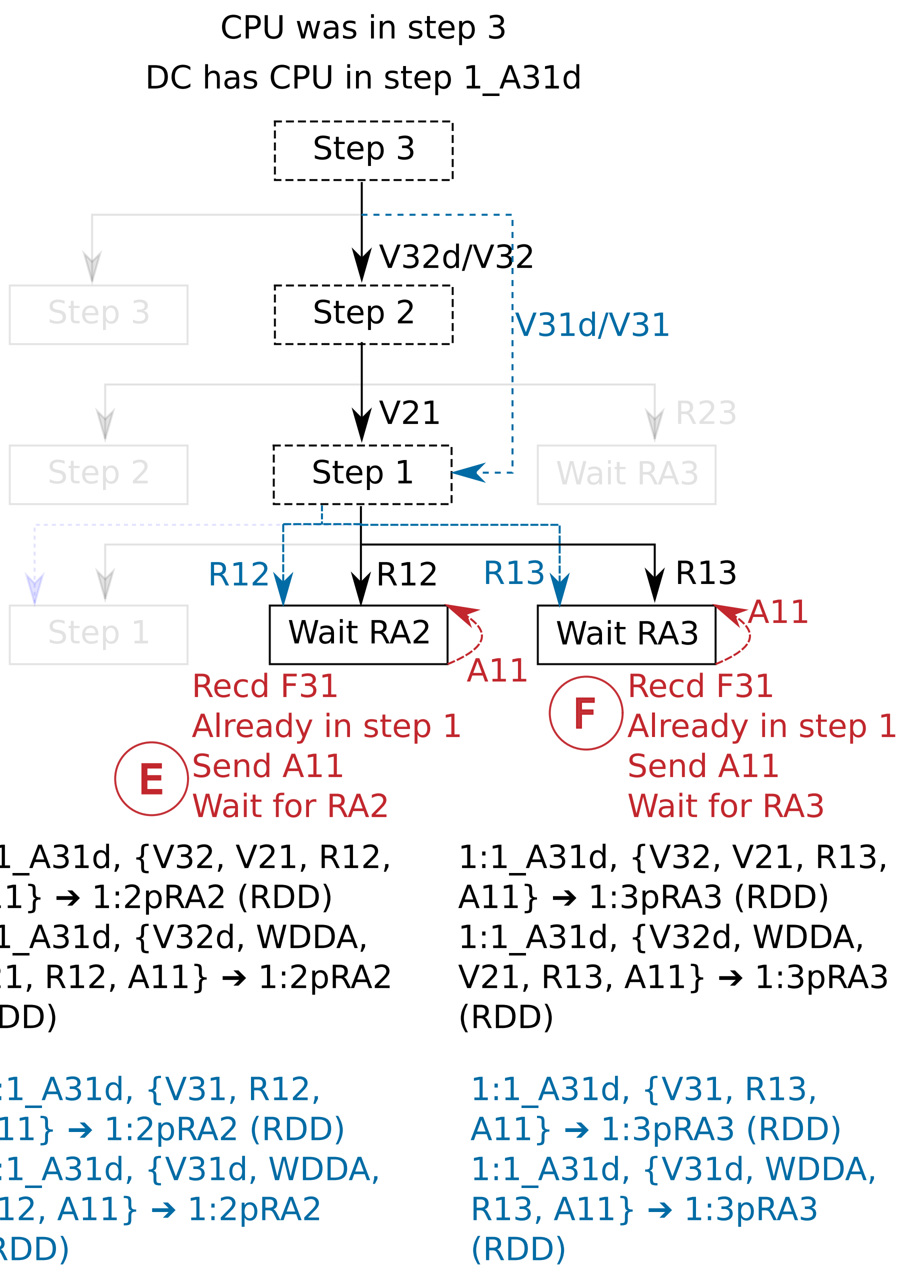

CPU was in step 3
DC has CPU in step 1_A31d
Step 3
V32d/V32
Step 3
Step 2
V31d/V31
V21
R23
Step 2
Step 1
Wait RA3
D
R12
R12
R13
R13
Recd F31
Step 1
A11
Wait RA2
Wait RA3
Already in step 1
Send A11
1:1_A31d, {V32, V21, A11} ➔ 1:1 (No action)
1:1_A31d, {V32d, WDDA, V21, A11} ➔ 1:1 (No action)
1:1_A31d, {V32, V21, A11, R12} ➔ 1:2pRA2 (RDD)
1:1_A31d, {V32d, WDDA, V21, A11, R12} ➔ 1:2pRA2 (RDD)
1:1_A31d, {V32, V21, A11, R13} ➔ 1:3pRA3 (RDD)
1:1_A31d, {V32d, WDDA, V21, A11, R13} ➔ 1:3pRA3 (RDD)
1:1_A31d, {V31, A11} ➔ 1:1 (No action)
1:1_A31d, {V31d, WDDA, A11} ➔ 1:1 (No action)
1:1_A31d, {V31, A11, R12} ➔ 1:2pRA2 (RDD)
1:1_A31d, {V31d, WDDA, A11, R12} ➔ 1:2pRA2 (RDD)
1:1_A31d, {V31, A11, R13} ➔ 1:3pRA3 (RDD)
1:1_A31d, {V31d, WDDA, A11, R13} ➔ 1:3pRA3 (RDD)

Figure A.4: *Pathways CPU can take when F31 is received, scenario D: The CPU has come down from step 3 to 1 either directly (V31d/V31) or through step 2 (V32d/V32 followed by V21). The CPU receives F31 at step 1, so it issues conflict response A11 and can continue remaining in step 1 or make upgrade requests to step 2 (R12) or step 3 (R13). The specification state equations are shown below each pathway.*



CPU was in step 3
DC has CPU in step 1_A31d
Step 3
V32d/V32
V31d/V31
Step 3
Step 2
V21
R23
Step 2
Step 1
Wait RA3
R12
R12
R13
R13
Step 1
Wait RA2
Wait RA3
A11
A11
Recd F31
Already in step 1
Send A11
Wait for RA2
E
F
Recd F31
Already in step 1
Send A11
Wait for RA3
1:1_A31d, {V32, V21, R12, A11} ➔ 1:2pRA2 (RDD)
1:1_A31d, {V32d, WDDA, V21, R12, A11} ➔ 1:2pRA2 (RDD)
1:1_A31d, {V32, V21, R13, A11} ➔ 1:3pRA3 (RDD)
1:1_A31d, {V32d, WDDA, V21, R13, A11} ➔ 1:3pRA3 (RDD)
1:1_A31d, {V31, R12, A11} ➔ 1:2pRA2 (RDD)
1:1_A31d, {V31d, WDDA, R12, A11} ➔ 1:2pRA2 (RDD)
1:1_A31d, {V31, R13, A11} ➔ 1:3pRA3 (RDD)
1:1_A31d, {V31d, WDDA, R13, A11} ➔ 1:3pRA3 (RDD)

Figure A.5: *Pathways CPU can take when F31 is received, scenario E and F: The CPU has come down from step 3 to 1 either directly (V31d/V31) or through step 2 (V32d/V32 followed by V21). Then the CPU has made upgrade requests R12 or R13 and receives F31 when waiting for a response. The CPU issues conflict response A11 and continues waiting for a response to the upgrade requests. The specification state equations are shown below each pathway.*

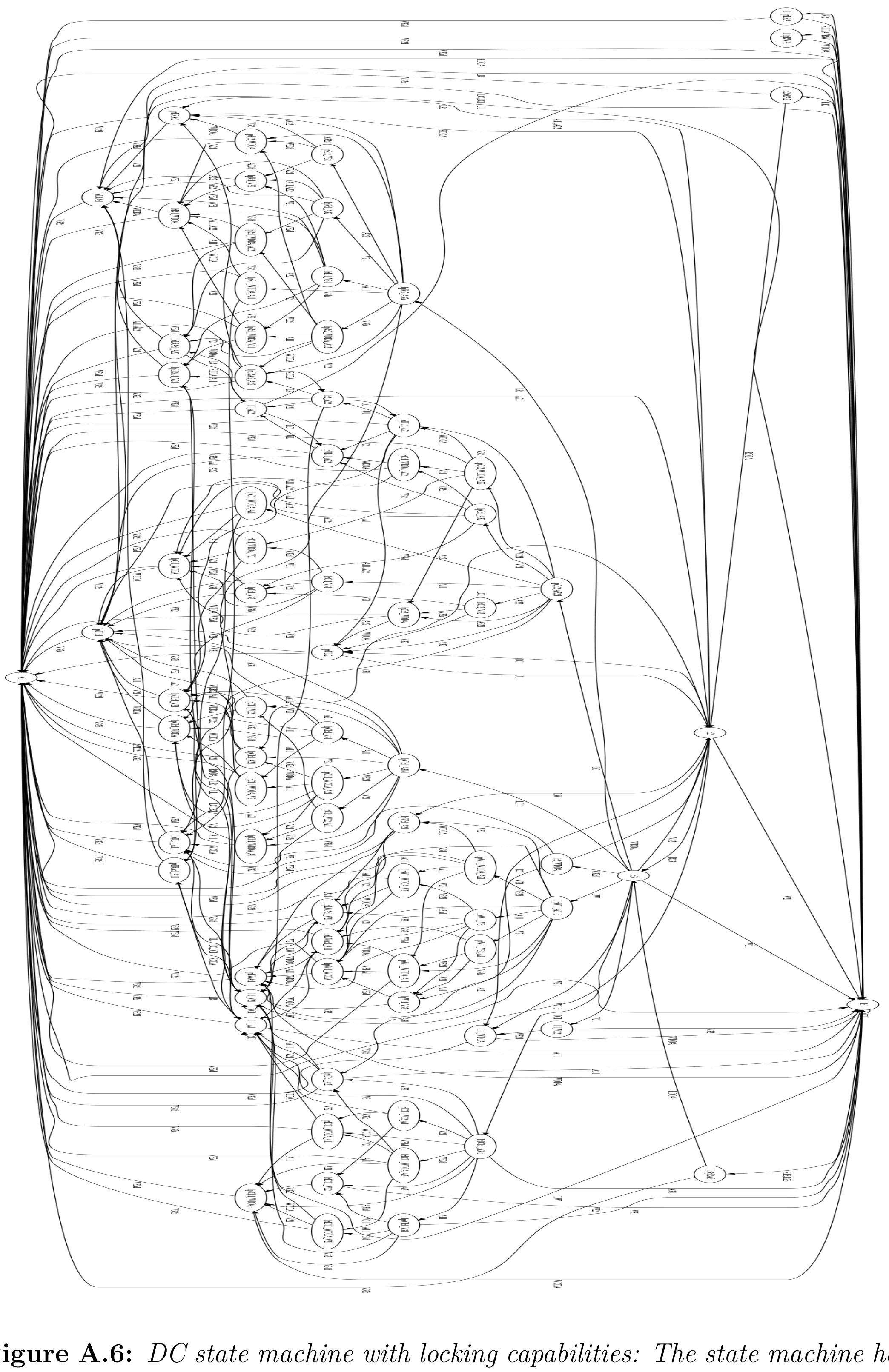


Figure A.6: *DC state machine with locking capabilities: The state machine has 79 states and 304 transitions and is automatically generated by the state space exploration tool. It shows that coherence protocols can be complex.*

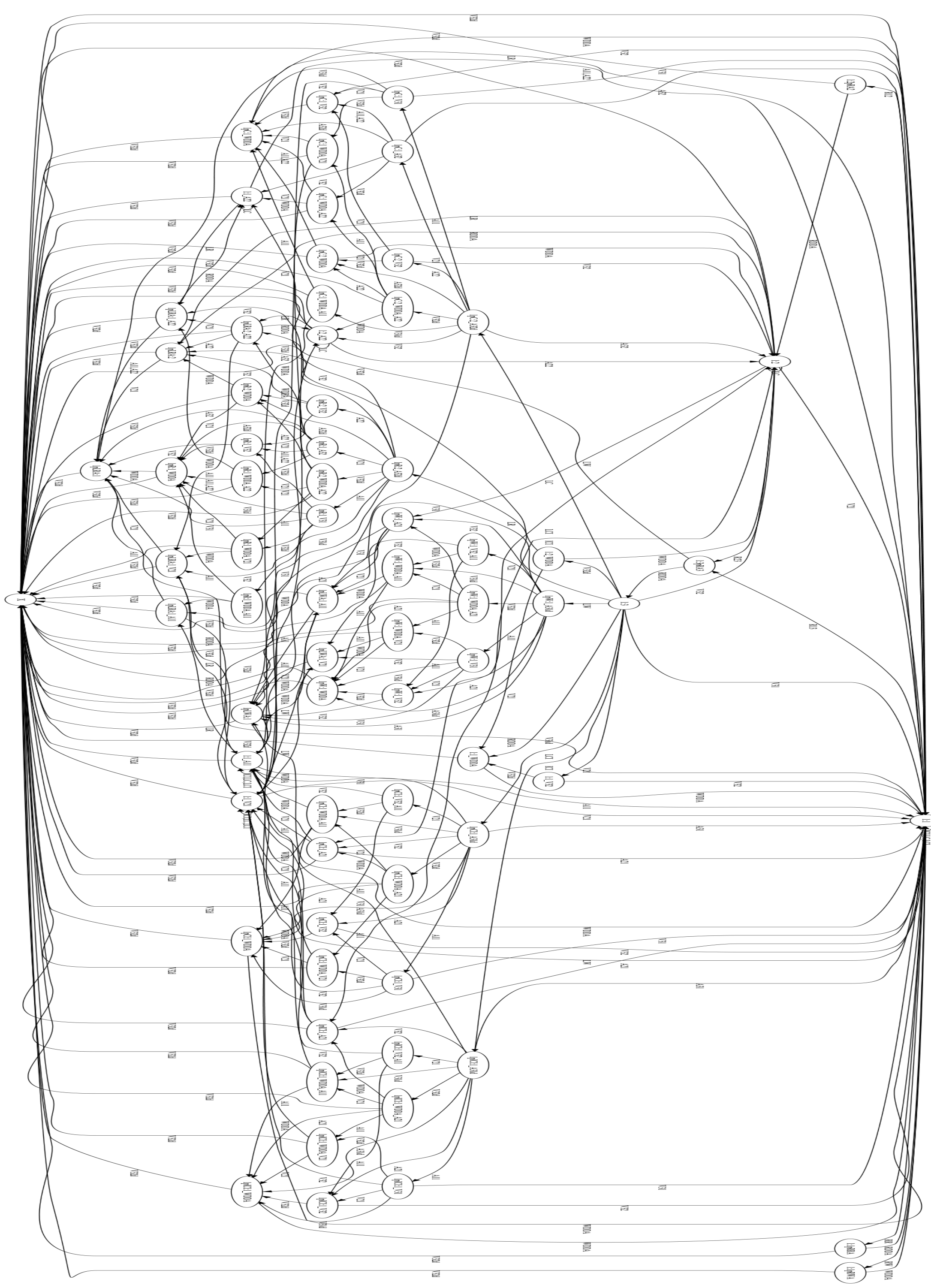


Figure A.7: *DC state machine without locking capabilities: The state machine has 73 states and 282 transitions and is automatically generated by the state space exploration tool. It shows that coherence protocols can be complex.*

DC Model	CCPI	Comment
LC	-	Local clean
LCI	-	Local clean invalidate
LR	-	Local read
LW	-	Local write
LCA	-	Local clean ack
LCIA	-	Local clean invalidate ack
LRA	-	Local read ack
LWA	-	Locak write ack
R12	RLDI/RLDD	Remote upgrade I to S
R13	RLDX	Remote upgrade I to E
R23	RC2DS	Remote upgrade S to E
RA2	PSHA	upgrade-ack S
RA3	PEMN	upgrade-ack E
V32	VICC.N (VICN)	Voluntary E to S no data.
V32d	VICC	Voluntary E to S with data.
V31	VICD.N	Voluntary E to I no data.
V31d	VICD	Voluntary E to I with data.
V21	VICS	Voluntary S to I no data.
F32	FLDRS_2H.E	Fwd E to S
F31	FEVX_2H.E	Fwd E to I
F21	SINV_2H	Fwd S to I
A32	HAKN_S	Fwd-ack E to S no data
A32d	HAKD	Fwd-ack E to S with data
A31	VICDHI.N	Fwd-ack E to I no data
A31d	VICDHI	Fwd-ack E to I with data
A21	HAKN (HAKD.N)	Fwd-ack S to I no data
A22	HAKS	Fwd-ack conflict S to S
A11	HAKV/HAKI	Fwd-ack conflict I to I
RR	RLDT	Remote read immediate
RW	RSTT	Remote write immediate
RRA	PSHA	Response to RR
RWA	PEMN	Response to RRA

Table A.1: *Mapping the names used for coherence messages in the DC protocol model to the names used in CCPI.*

B
Not-so-distributed Directory Controller

B.1 Introduction

This chapter discusses the design and performance of a preliminary version of the DC. In this version, the DC has only two DCUs. The directory of this DC is *not* sized to match the caching capacity of the CPU. With 8K directory entries, this DC allowed the CPU to cache only 1 MiB of FPGA cache lines. Moreover, this DC did not depend on the CPU to perform directory resource maintenance but rather self-maintained the directory using *Induced-Clean-Invalidate* transactions (section 7.9).

This DC also had two other differences when compared to the version of DC in chapter 8. First, each DCU could issue up to 64 on-going ECI, memory read and memory write transactions when compared to only one on-going transaction per DCU in the current version. The second difference is the presence of retry-buffers which stores coherence events that could not be handled. In the current version, whenever an event is not handled, it remains at the top of its channel and thus causes head-of-line blocking in that channel. With retry buffer, the un-handleable event is popped off its channel and pushed into the retry-buffer to be retried later.

B.1.1 Insights

Experimenting with this DC showed us what could be a major bottleneck in performance. The directory in the preliminary version of DC could hold the states of up to 8K cache lines. Once the CPU has cached 8K cache lines (1 MiB of data), the directory gets full and subsequent upgrades by the CPU would cause the DC to invalidate a previously cached cache line from the CPU's LLC to make space in its directory. Since each upgrade would require a round-trip invalidation, this becomes a major performance bottleneck. The solution to avoid these round-trip invalidations, the directory should be able to hold the states of as many cache lines that could be cached in the system.

Takeaway B.1. *The size of the directory matters limits the caching capacity in the system and also matters for performance. To avoid round-trip invalidations, the size of the directory should match the caching capacity in the system. If we assume that the CPU is the only node that can cache FPGA data and there are no caches in the FPGA, it makes sense to size the directory to match the 16 MiB caching capacity of the CPU.*

If we assume that each BRAM on the FPGA can store the state of 1K cache lines, we need 128K BRAMs to store the states of 128K cache lines (16 MiB of data). The FPGA has enough BRAM resources to accommodate for this [Xil21] but we still have two design choices to choose from.

First, we can continue having two DCUs per DC, with each DC having a directory that can hold 64K cache line states. This DCU would be highly pipelined, can issue 64 on-going transactions which can saturate ECI bandwidth and has retry buffers to reduce head-of-line blocking in incoming VCs. The problem with this approach is we need to place and route 64 BRAM units that communicate with a centralized DCU. This could lead to routing congestion and difficulty in meeting timing. Moreover the FPGA operates at a small frequency of 322MHz which can impact the latency of each DCU.

In the second option, we can have 128 DCUs with each DCU communicating with only 1 BRAM. This can simplify placement and routing of the directory but each DCU has to be extremely simple and optimized for resources for this to work. Since performance in the FPGA is achieved through parallelism this is the route we decided to go with.

In order to make each DCU as simple as possible, each DCU is not pipelined to get rid of resources that are required to track control hazards and scheduling of coherence events. Moreover, the resources that kept track of 64 on-going transactions are stripped to leave each DCU with enough resources to track only one on-going transaction. This is not a problem because we have 128 DCUs that can track 128 on-going transactions combined. Finally, the coherence events are routed to the cache lines through a routing logic with pipeline stages. These pipeline stages allow the coherence event at the top of a VC to be popped off as it gets routed to the DCUs. Thus these pipeline stages reduce head-of-line blocking, removing the need for additional retry buffers.

Takeaway B.2. *It is beneficial to have multiple parallel units of simple DCUs on the FPGA where each DCU is not pipelined, keeps track of one on-going coherence transaction and has pipeline stages at its input to reduce head-of-line blocking at the VCs.*

The rest of this chapter is organized as follows.

1. Section B.2 describes the architecture of the preliminary version of DC.
2. Section B.3 discusses the experiments that were run and highlights the performance bottlenecks in the design.

B.2 DC Architecture

The DC on the FPGA provides coherent access to FPGA-attached memory. It maintains a directory storing the home and remote states of every FPGA-homed cache line. Coherence controllers on CPU and FPGA exchange coherence messages with the DC at cache line granularity. Given a coherence message and the state of a cache line, the coherence protocol implemented by the DC dictates two things: First, what is the action to be performed and, second, what is the new cache line state once the action is performed. The DC performs four types of actions: **reading** from memory, **writing** to memory, **issuing** an outgoing coherence message, and **delaying** an incoming coherence message to be retried again later. After performing the protocol-specified action, the DC finishes by updating the directory with the new cache line state.

Although the CPU implements a full MOESI protocol, the DC on the FPGA is free to implement a subset of the protocol that is suitable to its needs. For example, the *owned* (O) state is required only in a three node system to allow inter-cache transfer of a dirty cache line between two remote nodes, bypassing the home node. Consequently, in a two node system, the FPGA can implement the MESI variant and still operate seamlessly with the CPU's protocol. Furthermore, the application's requirements can be used to fine-tune the protocol. For instance, in an application where only the CPU accesses the FPGA memory, the DC protocol does not have to support interactions with coherence controllers on the FPGA. This means that several variants of the protocol are possible and therefore it is beneficial to have a flexible architecture for the DC.

Any protocol variant can be represented as a table in which an event on a cache line and its present state can be looked up to get the next state and action. In spite of the hundreds of intermediate states to handle race conditions, the state transitions are always deterministic. Moreover the transition for a given cache line is independent of events and states of other cache lines. This implies the entire FPGA address space can be split across multiple *DC units* (DCUs), with each unit only responsible for a disjoint subset of cache lines that collectively cover the entire address space. This also presents an important design choice: A single, highly optimized DC for the entire FPGA address space or multiple simpler DCUs for exhaustive coverage of disjoint address spaces. In this paper we focus on the latter, a *distributed directory controller*, where each DCU is simple, un-pipelined, and optimized for resource utilization.

B.2.1 Directory Controller Unit (DCU)

In a distributed DC, coherence messages from multiple sources and virtual channels are routed to their corresponding DCU based on their cacheline address. In addition to coherence events, the DCU should also handle read and write responses from the byte-addressable memory. As a result, there can be events in multiple buffered channels waiting to be handled by a single unit at any given point in time. In line with keeping the design simple, the DCU chooses one event to handle at a time with a strict round-robin arbitration between all incoming channels. Although this choice simplifies the design, it can lead to deadlocks. For example, if the DCU stalls on an event due to unavailability of resources, any subsequent event that would free up this resource would also get stalled leading to a deadlock. To overcome this, the DCU must be able to switch to a different event in case the chosen event cannot be handled and, although not necessary, can choose to prioritize events that free up internal resources.

Whether the chosen event can be handled currently, depends on the protocol and availability of internal resources. In a simple un-pipelined design, the DCU can peek at the event at the head of the chosen channel to check if it can be handled. In case the event can

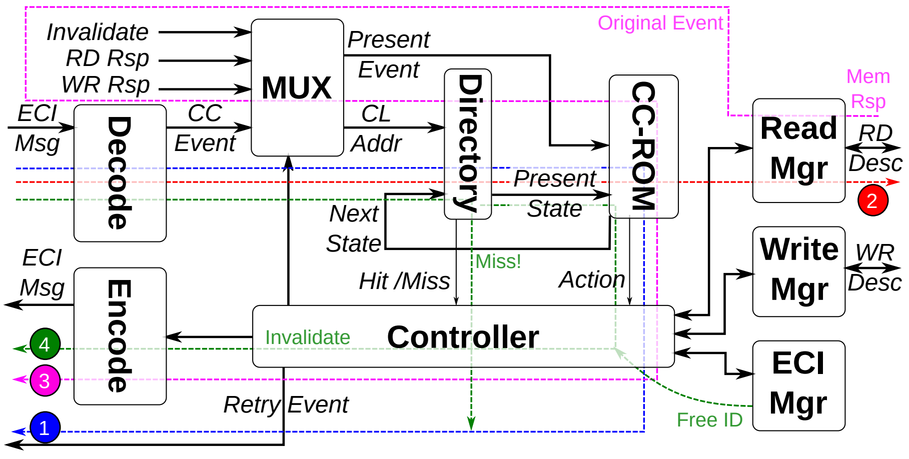


Figure B.1: *DC Unit Architecture*

be handled, it would be popped off its channel and the protocol-specified action would be performed. But if the event cannot be handled, the DCU can delay it by simply not popping it from its channel and choosing a different channel. One drawback of this approach is the head-of-line blocking on the ignored channel, which can affect performance especially in designs with fewer distributed units. This drawback can be mitigated by popping the event off its channel, even when it cannot be handled, and pushing it into a *retry buffer channel* to be tried again later. Bringing all this together, we can define the interfaces of a DCU. The current implementation has one interface to receive coherence messages, one interface to issue coherence messages, an interface each for read/write descriptors with independent request and response channels, and one interface to connect an optional retry buffer.

Expanding on the basic operation, the DCU must first decode the incoming event to extract, among other information, the cache line address. This cache line address is indexed into the directory to get the present states of the cache line. State information along with the decoded event is then found in the protocol table to get the next state and action. Finally the action is performed and the directory is updated with the cache line's new state. This gives us an idea of the components required: an event decoder, a directory to store and retrieve state information, a protocol look-up table, individual components for each action to be performed, and a controller that orchestrates everything. These components are shown in Figure B.1. Sorted by implementation complexity, the simplest component is the coherence protocol look-up table. It can be implemented as read-only memory since the protocol is fixed for the entire duration of the application, irrespective of

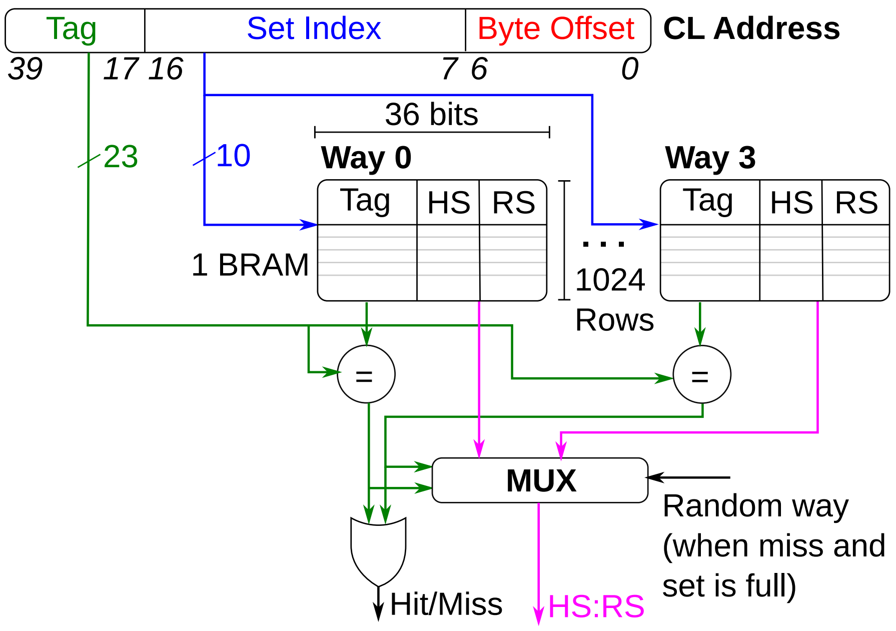


Figure B.2: *Directory Implementation*

protocol variant. Next is the event decoder which uses the ECI specification to decode the 64-bit header information in the coherence message to get its type, cache line address, size of payload, etc. The implementation of the directory and action handlers are described in the following sections.

B.2.2 Directory implementation

In a two node system, a cache line from home node can be cached either in home or remote or both. The state of a cache line in home and remote node is tracked in a directory. Ideally, the directory contains separate entries for all cache lines that belong to the home node but this consumes significant resources. One property that can be used to reduce the directory size is that, for a line not cached anywhere, both home and remote states default to invalid and need not be tracked. Although this reduces the size requirements of the directory, it can still be prohibitively large for a system with large caching capacity. In order to overcome this, we take advantage of the symmetric write-invalidate protocol. In a symmetric protocol, the home node has the ability to invalidate any of its cache lines that are cached anywhere in the system thereby limiting the total number of cacheable home lines. An invalidation can be done by the home node by issuing a downgrade-to-invalid coherence request to all caching units that have a copy of the cache line. Caching units invalidate their copies and acknowledge the request, at which point home marks the cache line as invalid in its directory. So, at the cost of extra communication and loss of locality, we can have a fixed-size directory that invalidates previously-cached cache lines to make space for new entries.

A fixed-size directory requires a placement policy for cache lines, the most general of which is n-way set-associative mapping shown in Figure B.2. Here, the directory can be viewed as a table with n columns (ways) and certain number of rows (sets). The set to which a cache line gets mapped depends on its address which is split into three fields: tag, set-index, and byte offset. After determining set-index, the cache line is placed in one of the available "free" ways by writing the state, tag information, and a valid bit to mark the way as occupied. If all ways are full, the replacement policy determines which cache line should be invalidated to create an empty spot. When reading the directory for a cache line, all n ways of the set are read in parallel and the cache line tag to be read is compared with the tags in the ways. A hit provides the present home and remote states of the cache line and a miss indicates that the cache line is not cached anywhere. Updating the state of an already existing cache line requires only writing the new state to the hit way.

In the current implementation, we have a 4-way set-associative directory where each way fits into 1 BRAM unit. In Ultrascale FPGAs, a unit of BRAM provides 36 Kb of storage and can be configured to have 1 K entries of 36-bit data. The 40-bit cacheline address is split into 10-bit set-index for 1K rows, 7-bit offset and remaining 23 bits are the tag. Of the 36-bit data entry, 23 bits would be occupied by the tag which leaves 13 bits for the home state, remote state and valid bit (sufficient for this application). In addition, a simple dual port configuration of the BRAM provides separate channels for reading and updating states making the design simpler. The DC implements an on-demand random replacement policy: When a set is full and a new entry needs to be added, a tag is chosen at random from the 4 ways and invalidated. This replacement policy was chosen with a sequential workload in mind and multiple invalidations can be batched together for better performance. To support the replacement policy, whenever the directory is read and there is a miss, in addition to indicating that the set it is full, the directory also returns tag and state information from a randomly selected way in this set, which can be used to trigger invalidations. Although not implemented in the current version, other replacement policies are possible (e.g. LRU).

B.2.3 Action handlers implementation

In lieu of describing each action handler separately, we discuss them in terms of the protocol pathways. These pathways are marked using numbers in Figure B.1. All pathways start by decoding the event, followed by reading the directory, and looking up the action to be performed in the protocol table. The pathways differ in the action specified and upon completion, most pathways update the directory. The simplest pathway is when an event must be delayed or retried ①. The action can be as simple as pushing the event into a retry buffer or switching to a different input channel. No updates to the directory are required. Next pathway complexity occurs when the only action is updating the directory. For example, when the DCU receives a downgrade-to-invalid with no payload, the DCU only updates the directory, completing the transaction.

Slightly more complex are pathways that require accessing byte-addressable memory, indicated by pathway ②. In this case the action generates read or write descriptors, stores the causal event for future needs, and updates the directory. These actions are performed by *transaction managers* (TMs). Read/Write TMs start a memory transaction by storing the original event in a transaction table and generating memory-access-descriptors from the decoded event. They decouple memory and coherence transactions by providing the transaction table index as an identifier to track the memory transaction downstream. Eventually when a memory response is received, the identifier can be used to extract the original event and free up the transaction table. Thus the only action to be performed by the DCU while issuing memory requests is to enable the appropriate TM and update its directory. But a TM might not always be ready to issue new memory transactions. For example, the transaction table might be full or the memory is not ready to accept new transactions. In this scenario, the DC delays the current event and switches to handling a different event. Handling memory responses is similar to handling any other event and, depending on the protocol, might require issuing a coherence message (pathway ③). The message header is generated from the message retrieved from the transaction table and the optional payload is obtained from the memory response itself. Since memory response events free up internal resources, they are prioritized over coherence events.

Finally, certain events require a new entry to be created in the directory. When the directory is not full, reading the directory after decoding the event would result in a miss and would return the index (set and way) of an empty spot. Once the protocol-prescribed action is performed, the new cache line state can be updated in the spot returned by the

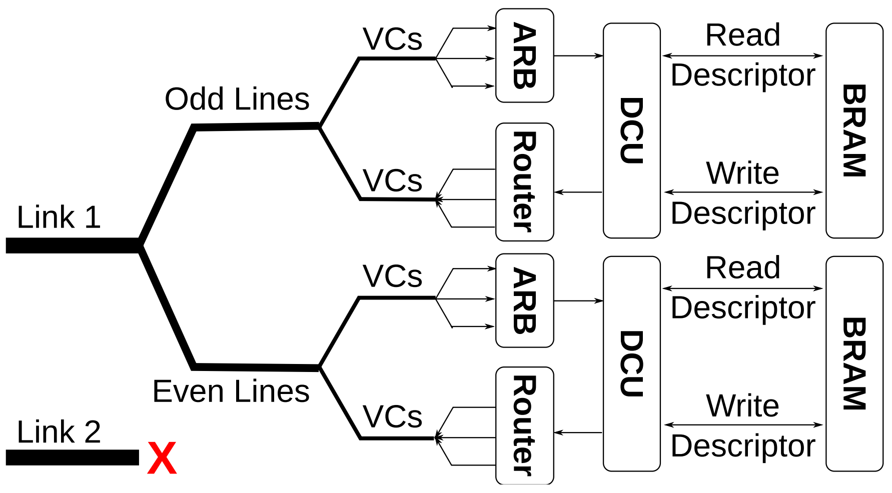


Figure B.3: *Experimental Design*

directory. But if the directory is full (pathway ④), it returns the state and tag information of a evictable cache line. Instead of continuing with the chosen event, the DCU performs resource maintenance by identifying the action required to invalidate the cache line returned by the directory. Once the action is performed, the invalidated cache line state is updated in the directory and the original event is delayed to be retried later. The action typically involves issuing a downgrade-to-invalid message, marked by a transaction identifier, to all coherence controllers that have a copy. These transaction identifiers are provided by a transaction manager which tracks ongoing coherence transactions issued by the DC. This transaction manager is similar the to read/write transaction manager except that it does not have to store the original event. In time, when the all downgrade acknowledgments from the coherence controllers are received, the cache line is invalidated and an empty spot is available. The performance of such a policy is affected by the invalidation process latency, thus multiple invalidations must be batched together to improve performance. For a given application, the number of batched invalidations depends on the number of DC units, the size of the retry buffer connected to each unit, and the number of on-going coherence requests that can be issued.

B.3 Performance and Resource Consumption

The design used in the DCU evaluation is shown in Figure B.3. Only one of two ECI links is used, capping the maximum theoretical performance to 15 GiB/s. Although the number

Retry Delay	In-flight Evicts	Throughput (GiB/s)		Latency (ns)
		Read	Write	
0	2	0.346	0.665	600
4	8	1.731	1.778	600
16	32	2.471	2.024	593
32	64	2.470	2.014	673

Table B.1: *Performance of distributed directory controller*

of DCUs is limited only by resource availability, in this paper, and for reasons of space, we cover an implementation with just two. The FPGA address space is split into odd and even cache line indices with a DCU each, providing coherent access to the disjoint address spaces. For the experiment, an application on the CPU performs sequential caching reads and writes to a block of FPGA memory. Consequently, the directory controller only implements a protocol subset allowing CPU-only coherent access to FPGA memory. To avoid caching locality effects in CPU's cache, the block size accessed by the CPU is twice its caching capacity.

The performance of a distributed directory controller depends on several factors. The first is the number of DCUs. Each DCU has a fixed directory size which limits the total amount of cacheable FPGA data. For example, the current configuration has 8 K directory entries which limits the total cacheable FPGA data in the CPU to 1 MiB. Trying to cache more data results in an invalidation round trip and performance becomes latency sensitive. One way of hiding this latency is to batch invalidation requests. This batch size depends on the retry buffer size and the number of DCUs. In the existing design with only two directory controllers, the retry buffer size plays an important role in performance because without it there can be head-of-line blocking and only two outstanding invalidations. A second factor that influences performance is the memory access latency and the number of possible on-going memory transactions. The transaction table size and the number of DCUs determine the total number of on-going memory transactions. By tuning these parameters, the memory access latency can be hidden. That said, in the current system memory access latency is lower than the DCU latency and thus the measurements indicate best-case performance. Finally, DCUs are not pipelined and can handle only one event at a time. This places a cap on single unit performance. Throughput can be improved by increasing the number of DCUs, so optimizing resource usage of each unit is important.

LUT	Reg	CARRY8	F7 MUX	F8 MUX	BRAM
0.13%	0.11%	0.003%	0.01%	0.006%	0.003%

Table B.2: *DCU Resource Utilization*

Read/write throughput and latency are measured by implementing the 2-unit distributed directory controller on Enzian FPGA running at 322 MHz. Each DCU has a 4 K entry directory. The retry buffer can be implemented as a FIFO but, to simplify the design, it is implemented as a register pipeline. The retry buffer pipeline depth specifies the number of cycles after which an event would be retried and also influences the number of in-flight invalidations. All transaction managers can issue 64 on-going transactions, though this has no effect on performance as the memory access latency is negligible. The steady state performance of two DCUs is shown in Table B.3.

The first observation is that batching invalidations together increases performance up to a saturation point, beyond which the delay between retries adversely affects latency. The second observation is that write performance is slightly lower than read performance due to event arbitration. In this scheme, memory responses are prioritized over coherence events, and read responses are given a higher priority than write responses. This asymmetry is only an artifact of the current implementation and can be changed by implementing a round-robin arbitration between read and write responses. Our third observation is that round-trip invalidations, even when batched together, play a significant role in the system design and performance.

Finally, Table B.3 shows that each DCU utilizes only a minuscule portion of the FPGA resources. This means that the DCU can be duplicated many times depending on performance requirements. Given the caching capacity for the current system is 16 MiB (128K cache lines), with a directory of 1K sets and 16 ways per DCU, only 8 DCUs are required to keep track of all the cachelines in CPU's LLC. Such a design would use about 5% of the available BRAM and avoid any conflict invalidations.

List of Tables

List of Figures

Bibliography

[Ama23] Amazon Web Services. "Amazon EC2 F1 Instances: Enable faster FPGA accelerator development and deployment in the cloud.", 2023.

[BBB+11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. "The Gem5 Simulator." *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 1âĂŞ7, 2011.

[Ber22] Berkeley Architecture Research. "TileLink.", 2022.

[BGP+19] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, and O. Mutlu. "CoNDA: Efficient Cache Coherence Support for near-Data Accelerators." In *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 629–642. 2019.

[BK22] A. Brouwer and M. Kerrisk. "mmap(2) – Linux manual page.", 2022.

[BKdV03] M. Bezem, J. W. Klop, and R. de Vrijer. *Term rewriting systems.* Cambridge University Press, 2003.

[BKP20] H. Brais, R. Kalayappan, and P. R. Panda. "A Survey of Cache Simulators." *ACM Comput. Surv.*, vol. 53, no. 1, 2020.

[BTP+22] A. Bhardwaj, T. Thornley, V. Pawar, R. Achermann, G. Zellweger, and R. Stutsman. "Cache-Coherent Accelerators for Persistent Memory Crash Consistency." In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, pp. 37–44. Association for Computing Machinery, New York, NY, USA, 2022.

[BWPN18] N. Beck, S. White, M. Paraschou, and S. Naffziger. "“Zeppelin”: An SoC for multichip architectures." In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 40–42. 2018.

[CCF+16] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. "A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms." In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE Press, 2016.

[CCI19] CCIX Consortium and others. "Cache Coherent Interconnect for Accelerators (CCIX).", 2019.

[CCP+16] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. "A Cloud-Scale Acceleration Architecture." In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[CF78] L. M. Censier and P. Feautrier. "A New Solution to Coherence Problems in Multicache Systems." *IEEE Transactions on Computers*, vol. C-27, no. 12, 1112–1118, 1978.

[CFO+18] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, M. Ghandi, D. Lo, S. Reinhardt, S. Alkalay, H. Angepat, D. Chiou, A. Forin, D. Burger, L. Woods, W. Gabriel, M. Haselman, and D. Zhang. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave." *IEEE Micro*, vol. 38, 8–20, 2018.

[CGKS20a] I. Calciu, J. Gandhi, A. Kolli, and P. Subrahmanyam. "Using cache coherent FPGAs to accelerate remote access.", 2020. US Patent US10761984B2, Filed 2018-07-27, Issued 2020-09-01.

[CGKS20b] I. Calciu, J. Gandhi, A. Kolli, and P. Subrahmanyam. "Using cache coherent FPGAs to track dirty cache lines.", 2020. Worldwide Patent WO2020023791A1, Filed 2018-07-25, Published 2020-01-30.

[CIP+21] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. "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, pp. 79–92. Association for Computing Machinery, New York, NY, USA, 2021.

[CLL+18] J. Choi, R. Lian, Z. Li, A. Canis, and J. Anderson. "Accelerating Memcached on AWS Cloud FPGAs." In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, HEART 2018. Association for Computing Machinery, New York, NY, USA, 2018.

[Cor09] I. Corporation. "An introduction to the Intel Quickpath Interconnect.", 2009.

[CPK+19] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam. "Project PBerry: FPGA Acceleration for Remote Memory." In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pp. 127–135. Association for Computing Machinery, New York, NY, USA, 2019.

[CRS+22] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe. "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, pp. 590–607. Association for Computing Machinery, New York, NY, USA, 2022.

[CTL17] H. Cook, W. Terpstra, and Y. Lee. "Diplomatic Design Patterns: A TileLink Case Study." In *First Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*. 2017.

[CXL20a] CXL Consortium. "Compute Express Link.", 2020.

[CXL20b] CXL Consortium. "CXL Webinar: Introduction to Compute Express Link (CXL).", 2020. See in particular slide 14, time 30:58.

[EKK+23] M. Emami, S. Kashani, K. Kamahori, M. S. Pourghannad, R. Raj, and J. R. Larus. "Manticore: Hardware-Accelerated RTL Simulation with Static Bulk-Synchronous Parallelism." *arXiv preprint arXiv:2301.09413*, 2023.

[FD17] D. Foley and J. Danskin. "Ultra-Performance Pascal GPU and NVLink Interconnect." *IEEE Micro*, vol. 37, no. 2, 7–17, 2017.

[FPM+18] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. "Azure Accelerated Networking: SmartNICs in the Public Cloud." In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pp. 51–64. USENIX Association, USA, 2018.

[Gen20] Gen-Z Consortium. "Gen-Z Core Specification 1.1.", 2020.

[GIG23] GIGA-BYTE Technology Co., Ltd. "R150-T61 rev. 110) 2U ARM Rackmount Server.", 2023.

[GSL+22] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. "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, pp. 417–433. Association for Computing Machinery, New York, NY, USA, 2022.

[HCW+19] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. "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. 2019.

[Int20] Intel. "Intel Agilex FPGA Product Brief.", 2020.

[JCLJ08] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. "CMP $im: A Pin-based on-the-fly multi-core cache simulator." In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pp. 28–36. 2008.

[JYP+17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao,

C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pp. 1–12. Association for Computing Machinery, New York, NY, USA, 2017.

[KCA92] J. Kubiatowicz, D. Chaiken, and A. Agarwal. "Closing the window of vulnerability in multiphase memory transactions." *ACM SIGPLAN Notices*, vol. 27, no. 9, 274–284, 1992.

[KLP$^+$18] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. "Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS." In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pp. 107–127. USENIX Association, USA, 2018.

[KRA20] D. Korolija, T. Roscoe, and G. Alonso. "Do OS abstractions make sense on FPGAs?" In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 991–1010. USENIX Association, 2020.

[LBH$^+$23] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. "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, pp. 574–587. Association for Computing Machinery, New York, NY, USA, 2023.

[LSC$^+$20] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker. “Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect.” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, 94–110, 2020.

[LXA$^+$21] N. Lazarev, S. Xiang, n. Adit, Z. Zahng, and C. Delimitrou. “Dagger: Efficient and Fast RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs.” In *ASPLOS*. 2021.

[LYT$^+$21] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee. “MIND: In-Network Memory Management for Disaggregated Data Centers.” In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP ’21, pp. 488–504. Association for Computing Machinery, New York, NY, USA, 2021.

[MHEH$^+$19] S. W. Min, S. Huang, M. El-Hadedy, J. Xiong, D. Chen, and W.-m. Hwu. “Analysis and Optimization of I/O Cache Coherency Strategies for SoC-FPGA Device.” In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 301–306. 2019.

[MWD$^+$23] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan. “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, pp. 742–755. Association for Computing Machinery, New York, NY, USA, 2023.

[MZL$^+$20] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci. “A Hypervisor for Shared-Memory FPGA Platforms.” In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’20, pp. 827–844. Association for Computing Machinery, New York, NY, USA, 2020.

[NBGS08] J. Nickolls, I. Buck, M. Garland, and K. Skadron. “Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?” *Queue*, vol. 6, no. 2, 40–53, 2008.

[NSHW20a] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence.* Synthesis Lectures on Computer Architecture. Springer CHAM, second ed., 2020.

[NSHW20b] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence: Second Edition.* Morgan and Claypool, 2020.

[OSC$^+$11] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta. "A Reconfigurable Computing System Based on a Cache-Coherent Fabric." In *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs*, RECONFIG '11, pp. 80–85. IEEE Computer Society, USA, 2011.

[PCC$^+$14] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services." *SIGARCH Comput. Archit. News*, vol. 42, no. 3, 13–24, 2014.

[PMR$^+$23] T. I. Papon, J. H. Mun, S. Roozkhosh, D. Hoornaert, A. Sanaullah, U. Drepper, R. Mancuso, and M. Athanassoulis. "Relational Fabric: Transparent Data Transformation." In *2023 IEEE International Conference on Data Engineering (ICDE).* Anaheim, California, USA, 2023.

[RG83] C. Ravishanicar and J. R. Goodman. "Cache implementation for multiple microprocessors." In *Proceedings of the 26th IEEE Computer Society International Conference (COMCON).* 1983.

[RSC$^+$21] P. Ranganathan, D. Stodolsky, J. Calow, J. Dorfman, M. Guevara, C. W. Smullen IV, A. Kuusela, R. Balasubramanian, S. Bhatia, P. Chauhan, A. Cheung, I. S. Chong, N. Dasharathi, J. Feng, B. Fosco, S. Foss, B. Gelb, S. J. Gwin, Y. Hase, D.-k. He, C. R. Ho, R. W. Huffman Jr., E. Indupalli, I. Jayaram, P. Kongetira, C. M. Kyaw, A. Laursen, Y. Li, F. Lou, K. A. Lucke, J. Maaninen, R. Macias, M. Mahony, D. A. Munday, S. Muroor,

N. Penukonda, E. Perkins-Argueta, D. Persaud, A. Ramirez, V.-M. Rautio, Y. Ripley, A. Salek, S. Sekar, S. N. Sokolov, R. Springer, D. Stark, M. Tan, M. S. Wachsler, A. C. Walton, D. A. Wickeraad, A. Wijaya, and H. K. Wu. "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*, pp. 600–615. Association for Computing Machinery, New York, NY, USA, 2021.

[RTLV19] M. Radi, W. W. Terpstra, P. Loewenstein, and D. Vucinic. "OmniXtend: Direct to Caches Over Commodity Fabric." In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pp. 59–62. 2019.

[SA22] D. D. Sharma and I. Agarwal. "Compute Express Link 3.0 Standard." *Tech. rep.*, 2022.

[SBJS15] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel. "CAPI: A Coherent Accelerator Processor Interface." *IBM Journal of Research and Development*, vol. 59, no. 1, 7:1–7:7, 2015.

[Sch23] J. Schult. "Characterization and validation of an in-silicon cache coherence protocol implementation." Master's thesis, 2023.

[SGS10] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." *Computing in science & engineering*, vol. 12, no. 3, 66, 2010.

[SK13] D. Sanchez and C. Kozyrakis. "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems." *SIGARCH Comput. Archit. News*, vol. 41, no. 3, 475âĂŞ486, 2013.

[SSI+18] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison. "IBM POWER9 Opens up a New Era of Acceleration Enablement: OpenCAPI." *IBM Journal of Research and Development*, vol. 62, no. 4–5, 8:1–8:8, 2018.

[SWC+20] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. "StRoM: Smart Remote Memory." In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. Association for Computing Machinery, New York, NY, USA, 2020.

[Tan76] C. K. Tang. “Cache System Design in the Tightly Coupled Multiprocessor System.” Association for Computing Machinery, New York, NY, USA, 1976.

[Ter17] W. W. Terpstra. “TileLink: A free and open-source, high-performance scalable cache-coherent fabric designed for RISC-V.” In *Proc. 7th RISC-V Workshop*. 2017.

[The23] The Enzian Project. “Enzian.”, 2023.

[TS21] N. C. Thompson and S. Spanuth. “The Decline of Computers as a General Purpose Technology.” *Commun. ACM*, vol. 64, no. 3, 64–72, 2021.

[TSK+22] S. Tamimi, F. Stock, A. Koch, A. Bernhardt, and I. Petrov. “An Evaluation of Using CCIX for Cache-Coherent Host-FPGA Interfacing.” In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2022.

[TSS88] C. Thacker, L. Stewart, and E. Satterthwaite. “Firefly: a multiprocessor workstation.” *IEEE Transactions on Computers*, vol. 37, no. 8, 909–920, 1988.

[Ver] Verilator. “Verilator RTL Simulator.”

[WMW+16] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe. “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. 2016.

[Xil21] Xilinx. “UltraScale Architecture and Product Data Sheet: Overview.”, 2021. DS890 (v4.0).

[Xil22a] Xilinx. “UltraScale Architecture-Based FPGAs Memory IP (PG150).” *Tech. rep.*, 2022.

[Xil22b] Xilinx. “Xilinx Zynq Ultrascale+ MPSoC.”, 2022.

[YHS+23] Y. Yuan, J. Huang, Y. Sun, T. Wang, J. Nelson, D. R. K. Ports, Y. Wang, R. Wang, C. Tai, and N. S. Kim. “Rambda: RDMA-driven Acceleration Framework for Memory-intensive Âţs-scale Datacenter Applications.” In

2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 499–515. 2023.

[ZAB+22] M. Zhao, N. Agarwal, A. Basant, B. Gedik, S. Pan, M. Ozdal, R. Komuravelli, J. Pan, T. Bao, H. Lu, S. Narayanan, J. Langman, K. Wilfong, H. Rastogi, C. Wu, C. Kozyrakis, and P. Pol. "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, pp. 1042–1057. Institute of Electrical and Electronics Engineers Inc., 2022.

[ZGK+21] J. Zuckerman, D. Giri, J. Kwon, P. Mantovani, and L. P. Carloni. "Cohmeleon: Learning-Based Orchestration of Accelerator Coherence in Heterogeneous SoCs." In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pp. 350–365. Association for Computing Machinery, New York, NY, USA, 2021.

[ZL20] Y. Zha and J. Li. "Virtualizing FPGAs in the Cloud." In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pp. 845–858. Association for Computing Machinery, New York, NY, USA, 2020.

[ZWC+20] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao, Z. Huang, and J. Sun. "FPGA-Accelerated Compactions for LSM-based Key-Value Store." In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*. 2020.

[ZXX+17] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda. "The Feniks FPGA Operating System for Cloud Computing." In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17. Association for Computing Machinery, New York, NY, USA, 2017.

Curricilum Vitae

Abishek Ramdas

Education

2018 - 2023	**Doctoral Candidate in Computer Science** ETH Zurich, Switzerland
2010 - 2013	**Master of Science in Computer Engineering** NYU Tandon School of Engineering, United States of America
2006 - 2010	**Bachelors in Electrical and Electronics Engineering** PSG College of Technology, India

Professional Experience

2018-2023 **Doctoral Candidate, Institute of Computing Platforms (Systems Group)**
ETH Zurich, Switzerland

- Advisor: Prof. Dr. Gustavo Alonso.
- Part of the team that built Enzian, an heterogeneous platform combining a server-class ARM CPU and an FPGA using a coherent interconnect.
- Developed an open, customizable cache coherency stack on the FPGA, bringing coherence to FPGA attached memory.

2013-2018 **Senior Engineer, Yield Engineering Team**
Qualcomm Communication Technologies, United States of America

- Developed and maintained a large volume diagnostics flow that supports diagnosis of failure logs from more than 20 projects across multiple technology nodes.
- Worked with multiple stakeholders and tool vendors to deploy new diagnostic methodologies.

2010-2013 **Researcher, Design for Excellence Lab**
NYU United States of America, United Arab Emirates
- Investigated and designed several DFT techniques to provide a controlled cost-quality trade-off.

Selected Publications

Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research in ASPLOS22.

Slack removal for enhanced reliability and trust in DTIS 2014.

Testing Chips With Spare Identical Cores in TCAD 2013.

Toggle-masking scheme for X-filtering in ETS 2012.