A fully-functional Cache Control Coprocessor for Enzian

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 487

Systems Group, Department of Computer Science, ETH Zurich

A fully-functional Cache Control Coprocessor for Enzian

by

Manuel Hässig

Supervised by

Ben Fiedler
Prof. Dr. Timothy Roscoe

October 2023 – May 2024

D INFK

Abstract

The Enzian research computer offers a unique platform for testing novel cache coherence protocols in hardware with its 48-core ThunderX CPU and FPGA connected over a coherent interconnect with the Enzian Coherent Interface (ECI). The C3 system is a toolkit for experimenting with novel features of cache coherence protocols on Enzian. It consists of a Cache Control Coprocessor (C3) on an isolated CPU core to fetch instructions from the C3 envoy on the FPGA to execute. Applications on the FPGA can issue C3 instructions to the C3 envoy which will then be executed by the C3 on the CPU. This namely enables injecting an FPGA-homed cache line into the CPU L2 cache and expressing prefetching patterns for CPU memory by the FPGA when it has knowledge of the CPU's execution, e.g. when the FPGA transfers data which needs to be handled by the CPU. Further, the C3 enables the FPGA to interrupt CPU cores using C3 instructions which result in software generated interrupts when executed by the C3. The C3 system is designed to be extensible such that it can be used to implement other features not implemented by the native cache coherence protocol to allow prototyping novel coherence features in hardware. The interface of C3 on the FPGA is designed to be similar to the interface of ECI to imitate a cache coherence protocol. The C3 is implemented as an out-of-tree Linux kernel module and provides notifications as an interface to CPU applications to let them know when an FPGA applications sent notification instructions the let them know that data has arrived for the application to consume.

Acknowledgements

First of all, I would like to thank my advisor Ben Fiedler for his support and guidance and many fruitful discussions which improved this thesis considerably. I would also like to thank Prof. Timothy Roscoe for the opportunity to work on this interesting but nonetheless challenging subject. Further, I would like to thank Pencheng Xu for being a more or less willing rubber duck and a very competent help in all things FPGA; Jasmin Schult for sharing her wealth of knowledge on the hidden secrets of the ThunderX caches and making sure that I do not sprint the marathon distance; Roman Meier for all his IT-support and thoughtful reviews; David Cock for the clarity provided by his insights; Adam Turowski for his support with puzzling FPGA problems; and Nicolas Tischler for his help with figuring out how to allocate more SGIs. I would also like to thank the Systems Groups as whole for being such a collaborative environment and for facilitating many interesting lunchtime discussions.

Lastly, I would like to thank Ibuprofen for getting me through the last three days before my deadline in spite of fevers and headaches.

Contents

Acronyms

ATF ARM Trusted Firmware 60, 62

AXI Advanced eXtensible Interface 22, 70–72, 75, 76, 78

BRAM block random access memory 28, 75, 76, 78

C3 Cache Control Coprocessor 8–12, 24–29, 34, 35, 38–58, 62, 65–74, 76, 78, 80–89

CCPI Cavium Coherent Processor Interconnect 17, 18, 21

CMI Coherent Memory Interconnect 17

DC directory controller 7, 21–23, 61

DCS directory controller slice 22, 28, 29, 33, 34, 46, 74–79, 83, 85, 86

DDIO Data Direct I/O 8, 10

DMA direct memory access 10, 24

DRAM dynamic random access memory 18, 19, 21, 22, 28, 30, 34, 35, 41, 43, 58, 59, 63, 65, 76

ECI Enzian Coherent Interconnect 7–10, 17, 20–26, 28, 63, 70, 72, 78, 85, 86, 88

FIFO first-in, first-out 26, 27, 70, 78

FPGA field programmable gate array 7–11, 16, 17, 20–22, 24–29, 33–38, 40–49, 52–56, 59, 61, 63, 65, 70–74, 76, 80, 83, 85–89

GIC generic interrupt controller 25, 48, 62

GPIO general purpose input/output 25

HDL hardware description language 76

I/O input/output 8, 10, 11, 18–20, 25, 26, 39–42, 59, 63, 70, 72, 87

ILA integrated logic analyzer 70, 78

IOB I/O-bridge 17, 19, 47

IP intellectual property 70, 76

IPC interprocess communication 41

IPI inter-processor interrupt 62

IRQ interrupt request 40, 62, 63, 65

ISA instruction set architecture 29, 33, 42

LCI local clean invalidate 22, 75, 76, 78

LCIA local clean invalidate acknowledgement 22, 75, 78

LLC last level cache 8, 10, 17

MUX multiplexer 27

NIC network interface card 8, 10, 11, 25, 55, 87

NOP no operation 19, 32, 34, 44, 66, 67, 71

NUMA non-uniform memory access 16, 59

RCU Read, Copy, Update 40, 58

REPL read-evaluate-print-loop 63, 64

RPC remote procedure call 8

SGI software generated interrupt 8, 27, 41, 47–49, 53, 55, 62, 63, 65, 66, 69, 89

SMP simultaneous multiprocessing 40, 62

TLB translation lookaside buffer 18, 30, 31, 34–38, 41, 61

UL unlock 22, 75, 85

UMP user-level message passing 10, 27, 41, 51–53

UND UMP-like Notification Delivery 51–53, 55, 68

URPC user-level RPC 10, 51

VC virtual channel 22, 23, 78, 85, 86

VM virtual machine 11

WNS worst negative slack 78

1 Introduction

Workloads on CPUs had enjoyed "free" speedups from the exponential increase of transistor count predicted by Moore's law [46] and the increase of processor frequencies and reduction of transistor with the same power consumption from Dennard scaling [18]. Consequently, chip manufacturers were able to deliver better performance by only optimizing CPUs. As Moore's law and Dennard scaling have ended [21] the overheads of general purpose CPUs [16] become less tolerable and the remaining way to get better performance is via domain specific hardware [28]. This drives an increasing heterogeneity in hardware [60]. However, memory dominates these accelerators [15]. In a sense, they move the memory transfer from inside the CPU to the interconnect between all processing elements. Unsurprisingly, these interconnects become the bottleneck for performance of accelerated systems [67, 27, 42].

The pressure for performance and features lead to the development of the more general proposals OpenCAPI [57], GenZ [30], and CCIX [9], which have merged into CXL [11], as well as the GPU specific NVIDIA NVLink [24], and AMD Infinity Fabric [7] and RISC-V TileLink [55]. All of these new interconnects offer coherent memory access in some shape or form. The ongoing specialization seems to also have moved coherent memory as an essential guarantee for multiprocessing out of the CPU.

In order to enable relevant systems research on heterogeneous systems in light of this paradigm shift the Systems Group developed the *Enzian research computer* [10]. Enzian is a two-socket asymmetric NUMA system with a 48-core ThunderX ARMv8 CPU on one socket and a Xilinx UltraScale+ field programmable gate array (FPGA) on the other. The two are connected with a coherent interconnect running Enzian Coherent Interconnect (ECI) [52], which implements the CPUs native cache coherence protocol. Additionally, CCKit [51] provides an implementation of a directory controller (DC) on the FPGA such that it participates as a home agent, making it a symmetric cache coherence protocol. Further, CCKit provides a low level interface with the cache coherence protocol to applications on the FPGA.

This makes Enzian an ideal platform for evaluating future cache coherence protocols on real hardware as it provides introspection down to the transport protocol of coherence messages. However, while being representative of the current state of the art in coherent interconnects ECI does not have all features which might be desirable in a coherence protocol, such as pushing a cache line from the FPGA in to the CPU L2 cache. The goal

of this work is thus to provide a comprehensive toolkit to enable additional features of a coherence protocol on Enzian.

Managing caches effectively is essential for getting good performance on modern computer systems. If we want to write a fast matrix-matrix-multiplication for instance, we need to take into account the dimensions, associativity, and replacement policy of each level of the CPUs cache hierarchy and block our data accordingly. Despite this need, caches on CPUs are intentionally transparent to software. As caches effectively hide most memory latency, leveraging them to the fullest for transferring data to and from accelerators could yield significant benefits.

Caches work best, when we know which memory will be accessed next and *prefetch* that memory into the cache ahead of the access so we can evade the not so compulsory miss. Applying this idea to the data transfer from accelerators, we notice that when transferring data back to the CPU the accelerator knows that the CPU will access the memory it transfers in the near future. Therefore, it would ideally transfer the data directly into the CPU's cache in a form of *push-prefetching* or direct cache injection. Because data movement is among the most expensive and energy-consuming tasks in a data center [14] and the latency sensitivity of short-lived workloads like serverless functions and remote procedure calls (RPCs), these workloads should benefit from push-prefetching as a feature in a coherence protocol. Further, some input/output (I/O) bound workloads using accelerated devices like a smart network interface card (NIC) profit from the reduced latency due to direct cache access, as demonstrated previously using Intel's Data Direct I/O (DDIO) technology [22]. Thus, the main additional feature we provide is direct cache access for the FPGA on Enzian.

We introduce the Cache Control Coprocessor (C3) which gets around the issue that ECI can only pull cache lines from a different node and not push them to another node. Thus, we need to send a message from the FPGA to the CPU to pull a cache line into its last level cache (LLC). For this, we create an instruction queue on the FPGA and have the CPU poll that queue continuously. In order to minimize latency, we isolate a CPU core to only poll the instruction queue and execute the instructions.

Now that we have a side channel for the FPGA to make the CPU execute things on its behalf, we can also implement other useful things. For instance, the FPGA can direct the CPU to issue a software generated interrupt (SGI). This is useful on Enzian as, at the time of writing, the workings of the interrupt lines between the FPGA and the CPU have not been deciphered yet. Hence, until these interrupt lines between the two nodes are brought up, C3 can be used as a workaround.

In this work, we design C3 as a flexible system other work can use to easily add features to Enzian not possible to implement properly or as a way to prototype new mechanisms. To that end C3 is extensible by design and makes few assumptions about applications using it. We apply the Enzianeering credo "if in doubt, overengineer" [10]. For the core

features direct cache access and interrupts, we investigate how these are accomplished best on Enzian. After implementing C3, we use simple applications to characterize the latency and throughput of direct cache access and draw conclusions as to how the system is best used in applications. Based on these conclusions, we devise a handful of different usages of our system for different application needs and show possible improvements.

1.1 Goals

Our main goal in this work is to create a reusable system component to enable new operations in communication from the FPGA to the CPU on Enzian in the form of the C3 system. The overarching goal is to be able to prototype coherent interconnects with a superset of the features of ECI on Enzian and enable experimentation with those novel features.

Initially, C3 should support push-prefetching of cache lines from the FPGA to the CPU L2 cache, some form of notification that data has been transferred, and the ability to interrupt CPU cores from the FPGA. In order to make push-prefetching over C3 useful, we aim to make the implementation of C3 have as low latency as possible.

To facilitate easy usage in the future, we provide guidelines and examples for applications.

1.2 Related Work

Coherence interconnect prototyping In recent years many new coherent interconnects have been introduced with CXL [11], NVIDIA NVLink [24], and AMD InfinityFabric [7], illustrating the considerable interest in coherent inter-chip data movement. However, these technologies are neither built for experimentation, extensibility, or introspection. While they are useful points in the design space, they do not allow exploring the design space of coherent interconnects. Much closer to that goal comes BlackParrot [49] with its BedRock cache coherence system [63]. BlackParrot can be implemented as a soft core on an FPGA based on the source code available on GitHub, which allows for experimentation, extensibility, and introspection. Even when taped out, the BedRock protocol in BlackParrot can be reconfigured with a firmware update as it is microprogrammed. For instance, it can be configured to use different subsets of the MOESIF states as protocol states of the coherence protocol. RISC-V TileLink [55] is also an open standard for a coherent interconnect which could lend itself to being extended due to its relative simplicity. If implemented on an FPGA, one could then also get introspection.

The Enzian research computer [10] with its implementation of ECI on the FPGA is designed for introspection to the lowest levels. However, the coherence protocol is constrained to that of the CPU, which is why this work is needed for Enzian to be extensible.

Hiding memory latency using cache prefetching is an old idea now commonly implemented in chips as hardware prefetchers and generally available on CPUs [36, 45, 8]. However, hiding latency by having a core push cache lines into another cores cache with push-prefetching or cache injection [35, 44] is much less common. Both ideas are similar in nature, but cache injection is producer-driven and fails if data is fetched to early evicting the applications working set, while prefetching is consumer-driven and fails if data is transferred too late [38]. Different approaches for push-prefetching in CPUs as surveyed by Byna et al. [8] include run-ahead execution, helper-thread-based prefetching, memory-side prefetching using near memory computation [66], and server-based prefetching [64] which is close in spirit to C3 as dedicates compute to the task of push-prefetching. The Stanford DASH multiprocessor [37] featured the *deliver* and *update-write* operations for applications that cannot issue a prefetch early enough and thus need push-prefetching. *Update-write* sends data to all cores that have the updated cache line cached and *deliver* sends data to a specified set of cores.

Push-prefetching seems to be more prevalent for accelerated computing where knowledge of future data transfers from one to the other node are leveraged, which is the realm where C3 will be used. In this context, the goal of using push-prefetching is to achieve lower data transfer latencies than using direct memory access (DMA) by avoiding reading data from main memory. For instance in proactively pushing data from the CPU to the GPU, so the GPU will mostly hit when it starts processing data [68]. There has also been considerable work on direct cache access for NICs [31, 33, 58]. The study by Farshin et al. [22] based on Intel's DDIO technology on Xeon server CPUs [32] shows that not all applications benefit equally from direct cache access and it has to be tuned specifically to each application. In particular, they showed experimentally that contention in the LLC between code/data and I/O is a problem and therefore the two need to be separated, which has been shown previously in simulation by Tang et al. [59]. Wang et al. [62] reverse-engineered and modelled DDIO in detail and found that the increasing complexity of the memory subsystem and I/O stacks make leveraging direct cache access to the fullest difficult. Further, they found that the increasing complexity of coherence protocols and the overhead of on-chip networks blur the distinction between a cache hit and a cache miss leading to diminishing returns. C3 aims to get around this penalty by using existing mechanisms in ThunderX.

Polling memory for achieving low-latency I/O in the same way as C3 is polling the FPGA for instructions is a well-known technique. However, many times the benefit comes from bypassing the kernel. Barrelfish [3] uses polling shared memory for user-level message passing (UMP), its user-level RPC (URPC) [5] based cross-core communication mechanism which achieves a latency in the order of hundreds of cycles for communication

across different NUMA systems. For Barrelfish, polling for a long time is unreasonable as it would block progress for the rest of the core, so it blocks after some time. However, systems for accelerating I/O to the CPU dedicate an entire core to polling. The Virtualization Polling Engine [41] uses dedicated polling threads on dedicated CPU cores to access devices and provide virtual access points over cache-coherent memory to virtual machines (VMs) instead of using interrupts, which significantly reduces the virtualization overhead. In networking DPDK [50] uses poll mode drivers to bypass the kernel network stacks to reduce latency. DPDK is used by Azure with its SmartNICs [23]. Andromeda [17] achieves its lowest TCP roundtrip latency using dedicated busy-polling CPUs, but achieves higher throughput at slightly lower latency using Intel's QuickData DMA Engines, which enable larger data copies. Shenango [48] dedicates a core to kernel-bypassing I/O with NICs which exposes packet queues to applications that are reallocated to cores on very short time frames in order to increase CPU efficiency. As an alternative to userspace polling on dedicated cores, Basu et al. propose compiler interrupts [2]. A compiler interrupt is a call to an interrupt handler inserted into a binary at compile time based on a configured interrupt interval. As a case-study, they replaced Shenango's dedicated I/O core with compiler interrupts and showed that they could increase CPU efficiency with comparable throughput and latency.

In most offloading scenarios, the CPU instructs the accelerator what it should execute and the accelerator tells the CPU when it is done. C3 inverts this relationship and has the FPGA issue instructions to execute on the CPU. This is not unique as Elis et al. [20] recently proposed a similar notification mechanism between GPUs and CPUs to achieve higher parallelism between the two. However, this is mainly focussed on synchronization. Closer to C3 are GPU-to-CPU callbacks [56] where the CPU polls pinned memory where the GPU writes a callback. A callback can be a system call, a memory transfer or some CPU compute. There are synchronous callbacks, where the issuing GPU thread spins until the request completes, and asynchronous callbacks where synchronization needs to happen some time after the request.

1.3 Following Along

Throughout this thesis, if you, dear reader, have access to an Enzian you can follow along and execute the experiments. Whenever this is the case you will find a box with the information needed to run the experiment. While the readme in each project should be self-explanatory, we will go through some general setup steps here.

First, all projects in this thesis need a newer Linux kernel than provided in the Enzian golden images. Everything was built using an image on a custom debug configuration based on Linux 6.7.1, but any 6.x kernel should work. You can obtain a copy of the image with the custom debug kernel using the following command on `enzian-gateway`:

```
emg copy-image mhaessig-6.7.1-enzian-debug $IMAGE_NAME
```

For general information on how to boot an Enzian, please refer to the Enzian Quickstart Guide. For the experiments, we boot an Enzian with core 47 isolated, which is where the C3 will be placed. Use the following command on `enzian-gateway` to boot an Enzian with core 47 isolated and the debug build of Linux 6.7.1:

```
emg acquire -n $IMAGE_NAME -k mhaessig/vmlinuz-6.7.1-enzian-debug \
    -i mhaessig/initrd.img-6.7.1-enzian-debug \
    -a "isolcpus=nohz,domain,managed_irq,47 nohz_full=47 rcu_nocbs=47 \
    rcu_nocb_poll" zuestoll$NR
```

Most experiments will need some specific bitstream programmed on the FPGA. Every experiment will specify the bitstream needed. All FPGA projects can be found in the Gitlab group for this thesis.

Once you have yourself a running Enzian, you need to build the C3 Kernel Modules. This "monorepo" contains all needed kernel modules, libraries and scripts. Build all its contents with `make`. All scripts you run to start an experiment must be run in the root directory of this repository.

With all this preparation, you should be ready for your first experiment.

***Following along:* Experiment 1.1: Welcome**

Required bitstream: None
Script: `experiments/welcome.sh`
What you will see: A welcoming text and some information about the core you are running on.

2 Background

This chapter will cover a range of topics used later in this thesis. The content covered in each topic is limited to what is relevant for this work.

2.1 CPU Caches

As the performance improvements of CPUs outpaced the memory subsystem, designers were forced to come up with a solution to the higher relative cost of memory access. The solution was to introduce a smaller but faster memory close to the CPU that stores blocks of memory the CPU accessed recently. This solution worked to increase the speed of memory accesses as the approach exploits the temporal and spatial locality inherent in programs, i.e. in loops accessing an array.

The simplest form of a cache is a *direct-mapped* cache which is basically an array of n blocks, called *sets*, of m bytes, which is the *line size*. To access the cache one takes the *index* of $\log_2(n)$ bits starting at bit $\log_2(m)$ of the address into the cache. If valid data is present in the specified cache set, the cache has to check if it is the same line as requested. It does this by comparing the *cache tag* stored in the cache and the most significant $64 - \log_2(n) - \log_2(m)$ bits of the address. If the tags match, the cache access is a *hit*. Otherwise, it is a *miss* and the cache replaces the line by evicting the cached line and fetching the requested line from main memory. To facilitate this functionality each cache line must store the tag, a valid bit, and the actual data. The hardware to check if the access into the cache is a hit in a direct-mapped cache is a comparator to compare the tag cached at the address-provided index with the tag of the address and an AND-gate to check if the valid bit is set if the tags match. This makes access incredibly easy and fast, but this simplicity comes at a cost. Direct-mapped caches encounter many *conflict misses*, where a cache line is evicted from the cache because a different cache line for the same set is requested.

When evicting a cache line, the cache has to decide which line to evict. This is also straightforward in a direct-mapped cache as there is only one option. Then the hardware must make sure that the contents of the cache line are written back to memory if the line is *dirty*, i.e. it has been modified since it was fetched. However, there are two ways of ensuring that changes in the cache are reflected in memory. A *write-through* cache writes changes to memory immediately, while a *write-back* cache waits until the line is evicted

to write the changes to memory. The trade-off between these two policies is that the former makes writes to the cache as slow as writing to memory but simplifies eviction, whereas the latter makes writes to the cache much faster but increases the miss penalty when a line has to be evicted.

In order to further exploit temporal locality and increase the size of a continuous working set, designers introduced *set-associative* caches where each set has a number of *ways* so multiple cache lines per set can be cached. For example a 2-way set-associative cache can store two cache lines of the same set before the request for a third cache line has to evict one of the two lines. On the opposite side on the set-associativity spectrum of a direct-mapped cache is a *fully-associative cache* which has one set and n ways, so an arbitrary memory location might be cached in any of the cache lines. The multitude of ways, however, makes the access much more complicated as the address-tag has to be compared with all tags in the cache set. This requires more hardware and more time and neatly demonstrates the trade-off for the set-associativity of a cache between hit rate and access time.

With multiple cache lines per set, the cache must pick a line to evict when all ways in a set are filled and a conflict miss occurs using a *replacement policy*. There are a host of different policies with varying levels of complexity. For instance, random replacement is quite simple as it needs no additional state stored in the cache line. There are simple queue-based policies with FIFO and LIFO. The least-recently-used policy and related policies evict the oldest line in the cache and needs to track the age of cache lines, which needs at least $\log_2$ bits for the number of ways for each cache line and logic to update the age bits. The family of frequency based policies uses the least-frequently used heuristic that needs to track the number of accesses per line and compare them on replacement.

The performance of replacement policies is analyzed by comparing them with the optimal replacement algorithm with perfect foresight, the clairvoyant Bélády's algorithm [4]. This policy replaces the line which will not be accessed again for the longest time in the future.

Since caches performance can be significantly improved by having foresight into what accesses will be made in the future, most caches implement some sort of *prefetching*. Some CPUs have hardware prefetchers that make predictions on future accesses and proactively fetch lines into the cache. CPUs also feature prefetching instructions which allow the programmer to inform the cache that some cache line will be needed in the future. However, in most architectures these prefetching instructions are implemented as performance hints and the CPU may ignore them entirely. For a comprehensive survey of prefetching techniques for CPU caches, see Mittal [45]. Further, Lee, Kim, and Vuduc [36] provide an analysis when prefetching is beneficial.

2.1.1 Cache Hierarchies

With time memories grew ever larger and with them the working sets of programs. Therefore, caches also had to grow. While a cache with low associativity will scale without too much increase in access latency, highly associative caches will not due to their need to compare cache tags in order to determine whether an access indeed hit. Faced with this dilemma, engineers determined that there will not be one cache to rule them all and introduced multilevel cache hierarchies. The first level (L1) cache closest to the CPU is rather small with access latencies in the order of three cycles. They often feature higher associativities to increase hit rates, which is acceptable due to their small size. The next higher level (L2) is bigger with access latencies in the order of tens of cycles. While there could theoretically be many more levels of caching, most cache hierarchies feature no more than three levels.

In von Neumann architectures, data and instructions are stored in one unified memory. Therefore, caches store both data and instructions. However, access patterns for the two are quite different. Instructions have quite good spatial locality as in code sections without jumps the next instruction is just executed (jumps can also relatively easily be prefetched using various lookahead techniques). Code with loops also has excellent temporal locality. Further, code size is bounded as it relates to the complexity of a program [19] These characteristics make code much easier to cache than code. For this reason designers introduced separate instruction and data caches in the lowest level of the cache hierarchy.

With more than one cache between the CPU and the backing memory not every cache line that is present in the L1 cache has to be present in the higher levels as well. This is a so-called *exclusive* cache. *Inclusive* caches, on the other hand, back all lines in the L1 cache with a line in the higher cache levels. While eviction is cheaper with inclusive caches as a line may still have room in a higher level, writing to a cache line in L1 can be faster in exclusive caches as they do not have to be propagated to the higher levels right away. Many caches implement a mixture between these two policies.

2.1.2 Cache Coherence

With the advent of multiprocessors the lower levels of the cache hierarchy were dedicated to a single core and thus replicated for every core and higher levels were shared between cores and connected using an interconnect. Now, every core may have different copies of the same cache line and read different values. To prevent such a scenario all cache controllers and memory controllers in the system cooperate using a *cache coherence protocol*. In short, a coherence protocol maintains two invariants [47]: for any memory location there exists exactly one core that may write to said location or there are a

number of cores reading said location at any given time; and after a write to some memory location, the value of that location will remain the same until another write occurs.

The cache controllers maintain a state machine for each cache line to uphold these invariants. Consider the MOESI variant of such a state machine. In the Modified state a core has modified the cache line and must thus be the only core holding a valid copy of the cache line. If a core holds a cache line in the Owned state, other cores may hold the line in shared state, but the owning core may invalidate their copies by writing to the cache line and transitioning to modified state. A cache line in Exclusive state is guaranteed to be the only valid and clean copy of that cache line in the system. The core holding it may read and write to the cache line as it pleases. A cache line in Shared state is a read only copy of some cache line owned by some other core. A cache line in Invalid state may be neither read from or written to.

To move between those states, the cache coherence protocol specifies a set of transactions which are initiated by coherence requests, e.g. a request to get a particular cache line in shared state. In snooping coherence protocols, all cache controllers observe coherence requests initiating transactions in the same order (per cache line) and then collectively take the correct action [47]. For this coherence request are broadcast to all cache controllers which causes a lot of traffic on the coherence interconnect. Directory based coherence protocols, however, keep track of the state of all cache lines in a global directory. This way, a coherence request needs to be sent only to the directory which decides on the coherence transaction [47]. This reduces the traffic on the bus dramatically, especially for large numbers of cores and thus cache controllers.

The concept of a coherence protocol can be extended to multi-node non-uniform memory access (NUMA) systems. However, this introduces a much more expensive inter-node interconnect into the system. In a *symmetric* distributed directory coherence protocol all nodes are equal partners and maintain a directory for the cache lines located in the memory of their own node, i.e. cache lines *homed* on the particular node. This is a common situation for multi-socket CPU systems [51]. For coherent heterogeneous systems however, not all non-CPU compute can maintain a directory for cache lines homed on its node. Thus, the CPU maintains a global directory across all nodes, making this an *asymmetric* directory coherence protocol. In such a system, the cache controllers of the node without a directory must send all coherence request across the coherence interconnect to the CPU on the other node.

2.2 Enzian

Enzian is an open CPU/FPGA platform for systems software research developed by the Systems group at ETH Zürich [10]. It consists of the powerful ARM CPU ThunderX-1 [43] on node 0 and a Xilinx XCVU9P UltraScale+ FPGA [65] on node 1 (see figure 2.1).

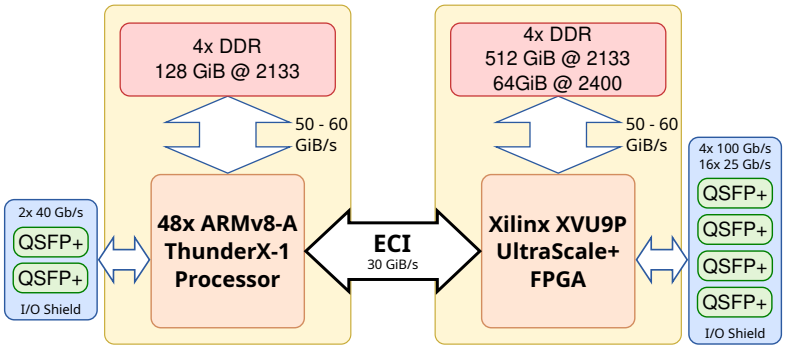

Figure 2.1: Enzian block diagram

The CPU and FPGA are both connected to their own set of DRAM. The main memory on both nodes is coherently connected by a high bandwidth interconnect running the Enzian Coherent Interconnect (ECI) protocol [52] which is based on the ThunderX Cavium Coherent Processor Interconnect (CCPI) protocol. This lets the FPGA participate in the cache coherence protocol of the CPU, which is one of the defining features of Enzian. Further, both nodes have ample network bandwidth — especially the FPGA.

2.2.1 ThunderX

The Cavium ThunderX-1 (see figure 2.2) is a 48-core server grade CPU which implements the ARMv8-A architecture and is clocked at 2.0 GHz [43]. It is itself a system on a chip (SoC) with a number of coprocessors for applications ranging from compression to finite automata. Each core features a 32 KiB, 32-way set-associative write-through L1 data (L1d) cache and a 78 KiB L1 instruction (L1i) cache. The ThunderX has a shared on chip L2 cache, which is also the LLC. The L1d cache and the L2 cache are both physically indexed and physically tagged while the L1i cache is virtually indexed and physically tagged. The cores, the L2 cache, the I/O-bridges (IOBs), and the CCPI are all connected coherently through the Coherent Memory Interconnect (CMI). It ensures coherent memory for all connected components by implementing a set of CMI transactions (see [43]) that uphold the coherence protocol. Some of these transactions are only initiated by prefetch instructions or the ThunderX specific Cavium cache management instructions (CvmCache) [43].

Each ThunderX core features 24 write buffers which perform aggressive write-combining to accumulate writes such that in the best case they can always write a full cache line into the L2 cache. The write buffers may stall the instruction pipeline if the no write buffer entries are available or when barrier instructions execute. The write buffers are flushed, among other events, when a barrier instruction that "must order prior stores to subsequent instructions executes" [43], the write buffer entry times out, or a load instruction to the same cache line as is being accumulated in that write buffer misses in

the L1d cache. Making sure that the write buffers are flushed to make writes visible to the rest of the CPU is critical for the correctness of some code.

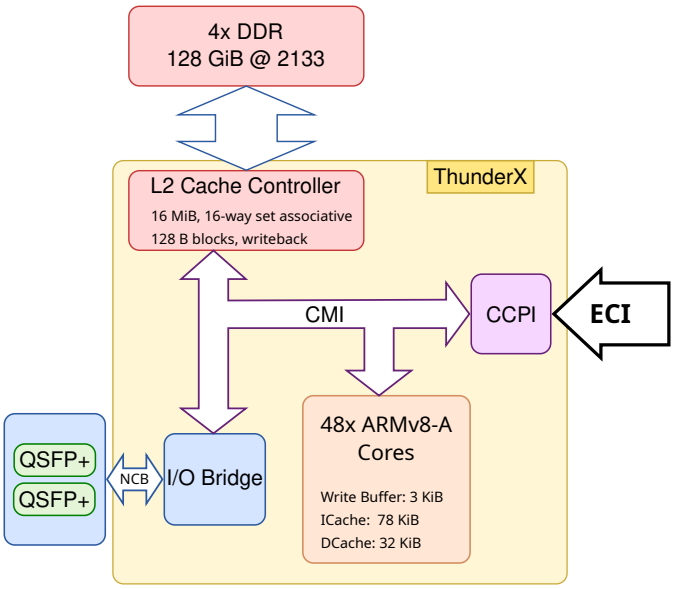

Figure 2.2: ThunderX block diagram

The ThunderX implements ARMv8 49-bit virtual addresses which are translated into 48-bit physical addresses using a two stage translation process over intermediate physical addresses. The physical address space is partitioned into dynamic random access memory (DRAM) and I/O address spaces. Only the DRAM address space is cached if caches are enabled. Bit 47 in the physical address distinguishes between the two: if address bit 47 is 1, then the address is in I/O space and in DRAM space otherwise. The DRAM and I/O spaces are further partitioned in to subspaces for each CCPI node (see [43] figure 4-1). Each CCPI node has a DRAM address space of 1 TiB.

Each ThunderX core features a hierarchy of three translation lookaside buffers (TLBs) to cache address translations. The 32-entry, fully-associative µTLB caches collapsed address mappings of both stages and is the TLB equivalent of an L1 cache. The MTLB is a larger, fully-associative, 256-entry TLB caching both stage 1 and stage 2 translations. It is the equivalent of an L2 cache. The last TLB in the hierarchy is the 128-entry, fully-associative page walker cache which caches translations for all page table levels except the last. Thus, it is a way to shorten page table walks for the page table walker. All TLBs on the ThunderX can be read out entry for entry using a corresponding CvmCache instruction. Further, the PREFu CvmCache instruction allows prefetching mappings into the µTLB.

Two ThunderX CPUs can be connected to a symmetric directory based cache coherent NUMA system thanks to the CCPI. It implements a reliable link layer protocol and three logical layer protocols. The CCPI logical coherent-memory protocol is a write-back, write-invalidate, home-based sparse-directory protocol that implements a MOESI

coherence protocol [52] for DRAM space. This protocol allows for out-of-order message transmission over the interconnect to minimize transmission latency and maximize usage of the interconnect. The other two logical protocols are used for I/O communication.

L2 Cache

The ThunderX L2 cache is 16 MiB large with 128 byte cache lines. It is 16-way set-associative write-back cache with 8192 sets. It can contain both cache lines from the home node and blocks from the remote node.

The replacement policy is implemented with a USED bit per cache way in each set. Generally, cache lines with an unset USED bit are considered for replacement. The USED bit is set to 1 when the L2 cache controller references a cache line. The L2 controller clears all USED bits but the bit in the last referenced block in all available ways in a set if all those USED bits where 1. If during a replacement operation all USED bits in all available ways of a set are 1, the first available cache line will be replaced [43].

L2 cache ways can be partitioned such that certain devices do not pollute the L2 cache. Way partitioning only affects cache replacement as it removes cache ways from consideration when performing replacement originating from specified cores or IOBs.

On the ThunderX individual L2 cache lines can be locked and are thus not replaced until they are unlocked. Cache lines can be locked using the CvmCache instruction fetch and lock which does what it says on the tin if there is an available way. Otherwise, it is a no operation (NOP). Cache lines can be unlocked with any explicit invalidation of the cache line, be it through an invalidation due to the coherence protocol or explicit invalidation using a CvmCache instruction.

CvmCache instructions implement address and index based variants to invalidate L2 cache lines. These instructions come in the Invalidate, Writeback, and Writeback and Invalidate flavors. Address based instructions identify cache blocks by physical address and index based instructions identify blocks by index and way. The latter are useful when flushing large portions of the L2 cache [43]. However, it is important to ensure that the cache index is calculated using the correct algorithm, i.e. mostly aliased.

The ARM architecture also features cache maintenance instructions (`dc` instructions) [39]. However, on the ThunderX all cache maintenance instructions except `dc zva` have no effect except that the core write buffer is flushed.

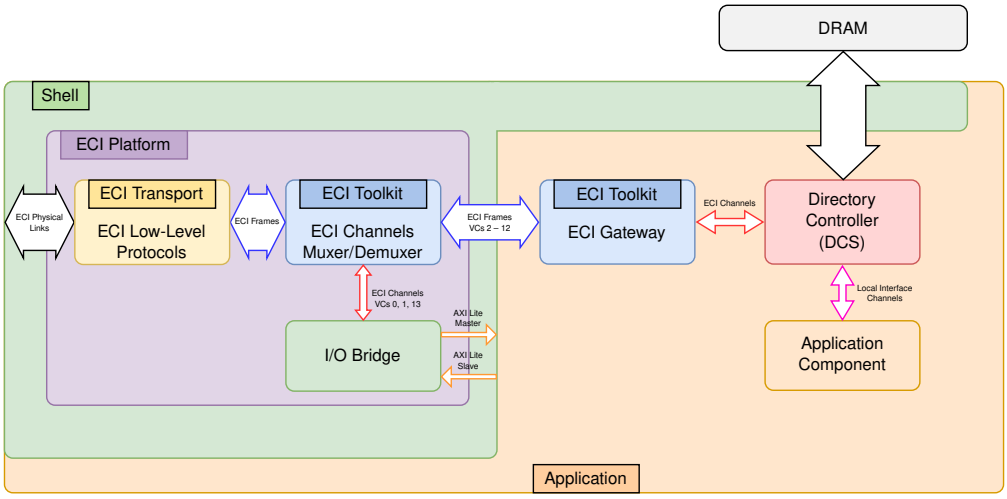

Figure 2.3: Block diagram of the Enzian Shell with a directory controller

2.2.2 FPGA and Shell

The Xilinx XVU9P UltraScale+ FPGA installed on node 1 in Enzian must be configured properly with an initial bitstream before it is able to communicate with the CPU on node 0. To that end Enzian uses an FPGA shell as shown in figure 2.3 akin to Coyote [34]. The Enzian shell[1] provides the basic ECI functionality such as link handling and ECI I/O message handling. Thus, the boot process of CPU must be paused using the Board Development Kit[2] until FPGA is configured properly. This is because the CPU attempts to bring up the ECI links during in the first steps of the boot process which will fail if the FPGA is not able to supply proper protocol replies [10].

Building an FPGA application for Enzian using the shell is a somewhat involved process. First, we need to build the shell with a stub application. The shell is configured such that it is placed in the static part of the FPGA the stub application is placed in the dynamically reconfigurable part. Then we can synthesize the application we actually want to build and link the synthesized shell with the synthesized application. This linked synthesized checkpoint can then be routed, placed and written to a bitstream, which can be programmed onto the FPGA.

[1] While there is no default shell at the time of writing, the static-shell seems to have established itself as the de-facto default.

[2] It is highly recommended to first get the CPU out of reset and then stopping the boot process using the BDK. The CPU sometimes exhibits problems when it is brought out of reset as there seems to be some reset timing issue between it and the FPGA. This exhibits as an exception during the boot process. To have a higher chance of not encountering an exception on the second boot attempt, the CPU should not be powered off, but rather be reset from the BMC console and its boot process again stopped using the BDK. After reprogramming the FPGA then, most of these boot exceptions go away (given the bitstream is fine).

The stub application[3] is interesting in its own right. It is pretty much the simplest application with which the CPU can communicate over ECI. As the "main feature", the stub provides two coherent cache lines in the DRAM address space. Whenever the CPU writes to the first cache line, the written value is copied to the second cache line and can subsequently be read there. This is the *cache line copy* often referred to later in this work. Both of these cache lines are controlled using a relatively simple single cache line controller. It implements the ECI transactions for reading and writing and exposes an interface to FPGA applications to perform coherent reads and writes to this cache line. For a full implementation of a directory, however, one needs to add the DC to an application.

2.2.3 Enzian Coherent Interface

2.3 Enzian Coherent Interconnect

The Enzian Coherent Interconnect (ECI) is an implementation of a cache coherence protocol in two-socket hybrid FPGA-CPU architectures. It is designed such that it implements the native ThunderX CCPI protocol and for implementation on an FPGA [52].

ECI, unlike CCPI, does not implement a full MOESI protocol. Instead, it implements an enhanced MESI protocol because it includes a transition (labeled 10 in figure 2.4 c) where the FPGA holds a dirty cache line in modified state and the CPU requests an upgrade of its copy of the cache line (currently in the invalid state) to shared, which is not included in plain MESI. But the inclusion of this transition in the protocol improves performance considerably as it avoids unnecessary writes to main memory and instead forwards the cache line [52].

ECI is an entirely pull based protocol. Meaning that a home node may not transfer data to a remote node without the latter first requesting said data. Therefore, the FPGA is not able to write data into the CPU's cache on its own initiative. Looking at all possible state transitions in ECI, there is no transition from a state where only one node has valid data to another state where either both nodes will have the same data or only the other node will have the same data. Moving data to another node is only possible via state transitions through the invalid state (see figure 2.4 c).

With coherence messages exposed to FPGA applications, these are able to participate in the coherence protocol and act like cache controllers making and serving coherence requests. This allows for some customization withing the bounds of the native cache coherence protocol and for manipulation of cache lines usually not possible with normal cache controllers.

[3]Link to repository: https://gitlab.inf.ethz.ch/project-openenzian/fpga-stack/dynamic-stub

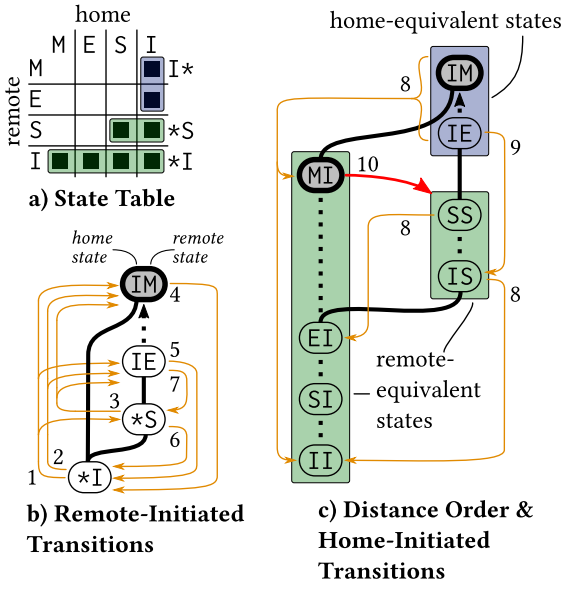

Figure 2.4: ECI protocol states and transitions as drawn by Ramdas et al. [52]

2.3.1 Directory Controller

The Enzian directory controller (DC) uses the coherence messages exposed by ECI to implement a symmetric directory coherence platform on Enzian [51]. The DC exposes 38 bits of coherent but uncached (on the FPGA) DRAM address space. It uses two parallel directory controller slices (DCSs), an odd directory controller slice (DCS) for handling ECI messages from odd numbered virtual channels (VCs) and managing even indexed cache lines and vice-versa. A DCS exposes an Advanced eXtensible Interface (AXI) interface to perform the reads and writes from and to some memory. The AXI interface is addressed using cache line indices shifted to the right by one bit, because this bit is the same for all indices of a slice. Further, a DCS exposes local VCs for local coherence requests.

Among the local coherence requests are the local clean invalidate (LCI) and unlock (UL) requests. The LCI request starts a coherence transaction which causes a writeback and subsequent invalidation of a specified cache line. Further, this locks the cache line such that the CPU cannot access it until it is unlocked. Once the LCI local coherence transaction is complete the DCS send a local clean invalidate acknowledgement (LCIA) message to acknowledge the completion. The UL coherence request simply unlocks the specified cache line. It does not have an acknowledgement and is assumed to complete immediately.

To use the DC in an application, we can include it as a submodule and add its sources in the project creation TCL script. We can then use the provided interfaces for coherent memory accesses after connecting the ECI VCs.

3 System Design

In this chapter we design the C3 system based on a select set of requirements. We design incrementally, taking fundamental design decisions which inform the further design. Based on a high-level design we investigate a few key mechanisms to inform the detailed design or the implementation or use of the system.

3.1 Design Requirements

The overarching goal of C3 is to provide additional features for ECI as a base for experimentation with future cache coherence protocols. To enable experimentation, we want C3 to be *modular* for easy use and *extensible* as we might not have foreseen all the needs of future research. Further, C3 should have as small of an impact on the rest of the system as possible.

However, we assume that applications on the CPU and their offloaded counterpart on the FPGA are co-designed and somehow have shared knowledge of some values. Further, we assume that the user controls the entire system. Therefore, C3 will not handle conflicting address spaces or identifiers between components.

Initially, C3 should implement the *injection of cache lines* from the FPGA to the CPU L2 cache, *interrupts of CPU cores from the FPGA* and *cache allocation*. In order to provide a good basis for research, cache injection should ideally be faster than DMA. Therefore, we require that C3 exhibits low latency so cache injection over C3 might have a benefit over plain old DMA.

3.2 C3 Design

The main challenge for C3 is that the operations the FPGA should be able to perform require actions on the CPU. Cache injection from the FPGA is not possible due to ECI only allowing nodes to pull data which therefor requires the CPU to request data when the FPGA pushes data. Interrupting CPU cores from the FPGA is currently mainly not possible on Enzian due to the interrupt lines in the serial link between the two sockets not

being reverse engineered yet, which is out of scope for this thesis[1]. Therefore, we need a communication channel between the FPGA and the CPU where the FPGA instructs the CPU to execute something. Further, this implies that we need a component on the FPGA to gather instructions and send them to the CPU and a component on the CPU to execute instructions.

In general, workloads on the FPGA are using it as an accelerator for some application managed from the CPU. The CPU might dispatch work to the FPGA and subsequently receive results or the FPGA might be doing some offloading on the I/O-path (e.g. a smart NIC) and the CPU will receive data without asking for it. With its initial feature set, C3 is only involved when the CPU is receiving data.

In order to get a prictoral overview over the design discussed in the following section, refer to figure **??**.

3.2.1 Control Path

Since we want to provide features additional to ECI the interface should not be too different. The interface to ECI is a ready/valid interface to submit packets on virtual channels. ECI needs to transfer packets that oftentimes contain data as it manages the transfer of data between nodes. The C3 does not directly transfer data, but only 64-bit instructions from the FPGA to the CPU, which may initiate a data transfer on the CPU. Thus, the interface for an application on the FPGA is a ready/valid interface to submit an instruction.

Design Decision 1. Applications on the FPGA use 64-bit instructions to delegate tasks to the CPU.

To transfer the instructions from the FPGA to the CPU we expose a register location in the I/O address space of the FPGA which the CPU can read to get an instruction. This leads to the question when the CPU should read from that instruction register. One option is for the FPGA to interrupt the CPU when a new instruction is ready. This is impractical as the only interrupts we can issue from the FPGA before this work are general purpose input/output (GPIO) interrupts and if two instructions arrive shortly after one another, the processing of the first instruction might be interrupted by the first. Another option is for the CPU to continuously poll for new instructions and execute them immediately upon receiving them, which naturally serializes the execution of C3 instructions on the CPU. Further, polled I/O is a known tool for reducing latency on data transfers [3] as data can be used immediately when it becomes available.

[1] Integrating the FPGA with the ThunderX's generic interrupt controller (GIC) is investigated by Nicolas Tischler at the time of writing.

However, polling continuously can severely detract from the performance of the entire systems as it consumes lots of CPU time spinning and it can also stress the memory bus with lots of read transactions. Since the ThunderX CPU on Enzian has 48 cores, we dedicate one of those cores to be the dedicated *Cache Control Coprocessor* to limit the adverse effects of polling to that core. Dedicating cores to polling I/O is also a well-known technique used by for instance by Andromeda [17], Shenango [48], and the DPDK [50]. To reduce the memory bus traffic from polling, the FPGA can stall the read while no instruction is available for at most the ECI timeout. Another advantage of isolating a core for the C3 is that it will be the only load on the core and thus have the L1 caches to itself. This should allow for less delay between subsequent fetches and thus lower latencies when multiple instructions are ready on the FPGA.

Design Decision 2. The Cache Control Coprocessor is implemented on an isolated CPU core to enable continuous polling for C3 instructions with reduced impact on the rest of the system and to enable a very tight fetch loop.

Because the C3 reads instructions from the FPGA at its own pace, we need some buffering between the application and the instruction read register. The *C3 envoy* provides the ready valid interface for applications to submit C3 instructions and the register for the C3 to read the submitted instructions. It provides buffering for instructions by implementing a first-in, first-out (FIFO) instruction queue, where the head of the queue is removed and returned when the C3 fetches an instruction. Further, it stalls the C3's read if the instruction queue is empty, as described above. If an instruction arrives during the stall, it returns that instruction immediately. However, if no instruction arrives before a set timeout, it returns a value signifying that no instruction was ready. Without this stalling mechanism the C3 would continuously fetch instructions in a very tight fetch loop which would generate loads read transactions and bus traffic. The aim of the read stalling is to reduce this traffic when it is not needed.

Design Decision 3. C3 instructions submitted by FPGA application are buffered by the C3 envoy which exposes a register location for the C3 to fetch the instructions from the queue. The C3 envoy stalls reads from the C3 when the instruction queue is empty to reduce bus traffic.

Note that the control path as described thus far decouples the submission of an instruction by a FPGA application from the instruction's execution on the C3. In other words, the application on the FPGA does not know at which point in time a C3 instruction it has issued is done executing. This makes it impossible for the application on the FPGA to notify its counterpart on the CPU that data is ready with "traditional" methods like ring buffers as this will race with the execution of a data transfer instruction. ECI is similar that the application cannot be sure when the transaction completes, but it knows when it issued the transaction. However, there is still a chance that packets might be reordered on the bus.

With C3, there are two straight forward ways to get around this. The first possibility is for the C3 envoy to signal the application when a particular instruction has completed, i.e. the CPU requested the next instruction after executing said instruction. This can be achieved by having the FPGA application execute in lock-step with the CPU or the submitted instructions get IDs which will be acknowledged by the C3 envoy after the C3 has executed them, which leads to some bookkeeping in the FPGA application to keep track of already executed instructions. The application can then ensure that "traditional" notification methods will be consistent. The second possibility is to make the notifications that data is ready for the CPU application to consume a C3 instruction themselves. Because instructions are submitted to a FIFO queue the C3 will execute them in order and one after another, so the data transfer and the accompanying notification are serialized and will not race anymore. For this work, we choose the second option due to its relative simplicity of piggybacking on the serialized execution.

Since we are executing notification instructions on the C3, we need to perform some cross core action to notify the application running on some different core. There are several ways to accomplish this: we could issue an SGI to the core running the application, we could implement some operating system primitives like `read()` or `poll()` which return when a new notification arrived, or we could implement some UMP-like transfer from the C3 to the application core.

Design Decision 4. The only explicit synchronization in the C3 system is the serialized execution of C3 instructions and notification instructions for FPGA applications to notify CPU applications. Since most applications need more synchronization, the design for C3-aware applications must include a synchronization contract between the CPU and FPGA applications.

Up to this point, we have only talked about a single application on the CPU and a single application on the FPGA working cooperatively. The design above can easily be adapted to multiple applications on both nodes. The C3 envoy only needs a multiplexer (MUX) on its ready/valid interface to accommodate multiple applications on the FPGA. The C3 itself needs a bit more work due to notification instructions. It needs to be able to map notifications to an app on the CPU. We enable this by having applications register with the C3 using an application id. The offloaded part of the application on the FPGA then needs to use this application id in its notification instructions. This setup theoretically allows for mapping many FPGA applications to one CPU application, but we disallow mapping one application id to multiple CPU applications. Sharing transferred data must be organized on the application level.

Design Decision 5. To enable multiple applications to use the C3, all CPU applications must register with the C3 using an application ID.

The C3 the CPU applications reside in different virtual address spaces and the FPGA does not have any address translation. However, the application on the FPGA must specify where the C3 needs to fetch data or which data is ready when notifying an CPU application. While it would be possible in principle to specify virtual addresses, it is much simpler and less error-prone to only deal in physical addresses. Since C3 instructions will mostly fetch from FPGA-homed memory, dealing in physical addresses is easier programs on the CPU will map the FPGA address space using `mmap(2)` and then compute addresses based on offsets from the base address. Further, it is also possible to identity map chunks of FPGA address space or even the entire address space, if needed.

To express more complex prefetching patterns, e.g. prefetch data needed for processing transferred data, the FPGA application needs to specify a CPU-homed address. In this case, the FPGA application needs to be informed of the locations of interest somehow (e.g. co-design with fixed locations, registers) and therefore it does not matter what kind of addresses it specifies as long as the CPU application knows how to interpret them correctly.

Design Decision 6. In general, addresses specified in C3 instructions are physical addresses. Deviations from this rule must be designed explicitly and require coordination between the CPU and FPGA application.

3.2.2 Data Path

One goal of the C3 is to offer data transfer from the FPGA to the CPU via direct cache injection. The only way to move data into the ThunderX's L2 cache is to have the ThunderX request it through a prefetch or a normal load as ECI only allows data to be pulled by a node. For the C3 to be able to fetch a cache line, the application on the FPGA needs to make it available over the interconnect. Uniquely on Enzian, this can be achieved by listening to cache coherence requests over ECI and respond to read requests on the cache line the application wanted to fetch directly with a response packet containing the data. The other way is to coherently write the data to be transferred to FPGA memory in the DRAM address space using the DCS. Interacting with ECI directly has the advantage that the application can issue a fetch C3 instruction as soon as the data is ready without having to write to memory beforehand. On the other hand it is much harder to implement correctly as we would have to deal with arbitrary coherence requests instead of interacting with the DCS to coherently write to memory. Further, we can reduce the latency of writing to memory by not writing to DRAM but using a smaller and much faster block random access memory (BRAM) dedicated solely to transferring data. Using crossbars, we can still use the rest of the address space for DRAM.

Design Decision 7. C3-aware applications use coherent writes to FPGA-homed memory to provide data that should be fetched by the C3.

In order to increase throughput and reduce the amount of instructions an application has to issue to transfer multiple cache lines, we want to be able to fetch multiple consecutive cache lines with one simple instruction. The largest transfer we want to allow, however, is 1 MiB, which corresponds to one way of the ThunderX's L2 cache. This is to limit the impact of C3 on the other users of the shared cache. With 1 MiB of data to transfer our approach of interacting directly with ECI will also need a sizable buffer, so we might as well write coherently to some scratch memory and be relieved of tracking cache line states. Thus and due to its relative simplicity, we choose the latter approach of coherently writing data to memory using the DCS before a data transfer.

Design Decision 8. To reduce the number of instructions needed and increase througput C3 instructions operating on cache lines should be ranged such that the same operation on multiple consecutive cache lines can be executed using a single instruction. The range in C3 instructions is limited to 1 MiB.

Once the C3 receives an instruction to fetch cache lines it can load the data from the scratch memory in the FPGA's address space as indicated in the instruction. We will investigate what the best method is for transferring a cache line into the ThunderX L2 cache in section 3.3.1.

An application can use data once it gets a notification from the C3 that some range of cache lines is now ready to be consumed. At this point the application can normally access memory at the specified location.

3.3 Key Mechanisms

In this section, we investigate some key mechanisms in detail to inform the detailed design of C3.

3.3.1 Direct Cache Injection

The C3 needs some way to transfer cache lines from the FPGA into its L2 cache. Performing a load on a word in a FPGA-homed cache line from the CPU will always fetch the line into the CPUs L2 cache. But this will also pollute the L1d cache of the core performing the load. Since we want the fetch loop of the C3 to be as quick as possible, we want to avoid polluting its L1 caches and thus avoid conflict misses.

To avoid polluting the L1 caches, we can use some form of prefetching into the L2 cache. The ARMv8 instruction set architecture (ISA) offers the `prfm p{ld,st}l2{keep,strm}` instructions. Further, the ThunderX offers a CvmCache system instruction L2 Fetch & Lock.

Prefetching on the ThunderX

In the ARMv8 microarchitecture prefetch instructions are considered performance hints and may be ignored by the processor [39]. On the ThunderX, testing by Jasmin Schult during her master thesis [54] suggests that prefetching will only occur if the prefetch would not incur a page fault. However, it is not clear exactly if this "page fault" is a TLB miss or some other fault in the address translation system.

To find out when exactly a ThunderX will indeed issue a prefetch, we have to get a closer look at the contents of its TLB just before a prefetch is issued. Luckily, the ThunderX provides system instructions for reading the contents of its TLBs [43] which we can utilize using the infrastructure described in section 4.1. We use the Enzian memory explorer described in section 4.1.4 to script different experiments to determine the prefetching behavior of the ThunderX.

To investigate prefetching on the CPU, we need to control a cache line in the L2 cache and write it back to DRAM so we can actually prefetch it. This does not sound too hard as we could simply write to some cache line and write it back. But doing so would only modify the cache line in the L1d cache of the core that wrote it and not immediately write through to L2 cache. The ThunderX implements an aggressive write-buffer that combines writes until a full cache line can be written to the L2 cache [43]. Before the write-buffer is flushed other cores can already access the modified cache line as the point of coherence on the ThunderX is in the L1d cache of every core. For our purpose of testing however, we want to ensure that all prefetches come from main memory and not from a write-buffer. On the ThunderX the write-buffer is flushed before a full cache line has been gathered when `dmb` or `dsb` barriers enforce ordering of stores, a load or prefetch instruction to the same cache line misses in the L1d cache, but hits in the write-buffer (i.e. another core accesses the modified subcacheline), and on architectural events such as a write-buffer entry timeout, more than a threshold of write-buffer entries are filled or a GlobalSync from outside the core arrives. In experiment 3.1 we see that the write-buffer entry timeout is at 2^{13} cycles. Thus, every write will be written to the L2 cache after at most $2^{13} \cdot 2\,\text{GHz} = 2^{-14}\,\text{s} \approx 61\mu\text{s}$. The ThunderX also implements all data cache maintenance instructions except `dc zva`[2] only as a write-buffer flush, since they are performance hints. Thus, the easiest way to ensure that a write ends up in the L2 cache for testing is to put a barrier after modifying cache lines.

[2] Zero out a cache line

***Following along:* Experiment 3.1: Memory Configuration**

Required bitstream: None
Script: `experiments/mem-conf.sh`
What you will see: The script shows the contents of the system register `CVMMEMCTL0_EL1`. Towards the bottom you can find the values regarding the write-buffer.

Writing back a cache line in the L2 cache to main memory is easy on the ThunderX as it implements special instructions for very specific cache management (see section 2.2.1). In this case we need CvmCache L2 Hit Writeback Invalidate, which writes back and invalidates a cache line based on a physical address if that address hits in the L2 cache.

Armed with the knowledge how we can control a cache line, we show that the ThunderX does execute a prefetching instruction when the mapping of virtual address of the cache line we want to prefetch is cached in the TLB and no page table walk is necessary.

As seen in section 2.2.1, the ThunderX has a translation hierarchy of multiple TLBs: the 32-entry µTLB and the 256-entry MTLB both caching virtual address mappings, and the 128-entry walker cache, which caches address translations for each level of the page table except the last. Hence, if a virtual address hits in the µTLB or the MTLB, no page table walk is required whereas if the address hits in the walker cache, the page table walker must still perform the page table lookup for the last level of translation.

On the ThunderX we can also inspect the contents of the three TLBs. Using the respective CvmCache instructions, we can read a single entry from one of the three TLBs into the memory mapped registers. By reading each entry of a TLB and comparing with a given virtual address, we can thus determine if this virtual address hits in the given TLB. For the implementation details refer to section 4.1.3. However, this introspection overhead of iterating over all entries actually makes it difficult to observe if a virtual address hits in the µTLB. Due to its small size and the large overhead, a virtual address often gets evicted from the µTLBby merely trying to observe it. Experiment 4.1 demonstrates this phenomenon.

With experiment 3.2 we see that the ThunderX only prefetches when the access does not need a page table walk. To show this we use a writing core c_1 and a reading core c_2. c_1 prepares two adjacent cache lines l_1, l_2 on the same page p and one cache line l_3 on the next page q such that they are flushed from the write-buffer and subsequently writes back and invalidates all of them. By accessing l_1, c_2 ensures that the mapping for p is valid in at least the MTLB which in turn ensures that prefetching l_2 succeeds. Since the reading core has not accessed a virtual address on page q, its mapping is not present in any of the core's TLBs. When c_2 prefetches l_3, the ThunderX will not execute it because a page table walk is necessary.

***Following along:* Experiment 3.2: Prefetching CPU-homed addresses**

Required bitstream: None
Script: `experiment/prefetch-homed.sh`
What you will see: Two tmux panes will open. Each with processes pinned to a different core. The core in the left pane writes to cache lines and the core in the right pane accesses cache lines. All addresses are physical addresses in the CPU DRAM address space.
First, prefetching a cache line on a page with its mapping in the MTLB or uTLB will succeed. Then, prefetching a cache line on a page without a cached mapping will fail.

While we cannot inspect the contents of the walker cache because we do not know how to interpret its contents, experiment 3.2 was set up such that the walker cache should contain the translation for the second to last level for l_3. We set up l_3 on page q that shares its second to last level translation table entry with page p, which contains l_2. Because we accessed l_2 right before we try to prefetch l_3 and the mapping for l_2 was cached in the MTLB, the shared second to last level translation table is most likely cached in the walker cache at the time of accessing l_3. This suggests that the ThunderX will not execute a prefetch if this access would cause a page walk for a single level.

Until now, we have used a load in an adjacent cache line to make sure that the address mapping is cached in the TLB. The ThunderX, however, implements a specific CvmCache instruction to prefetch an address mapping into the µTLB. According to the manual, the PREFu instruction prefetches the translation for the provided virtual address from the MTLB into the µTLB or be a NOP if the translation is not cached in the MTLB. Experiment 3.3, though, shows that PREFu will prefetch address translations which are not cached in the MTLB and that a prefetch will succeed afterwards. This suggests that we can use a PREFu-prefetch idiom to reliably prefetch cache lines using the usual ARM prefetch instructions without having to incurr an access on the same page beforehand. An alternative (unconfirmed) possibility is that the MTLB still has the stage 2 translation from intermediate physical address to the physical address store and the prefetch of the mapping succeeds because of this.

***Following along:* Experiment 3.3: Prefetching with PREFu**

Required bitstream: None
Script: `experiment/prefetch-prefu.sh`
What you will see: Two tmux panes with a writing core on the left side and a reading core on the right will open. First, the writing core writes to a cache line and writes it back to memory and invalidates the cache line. The reading core then uses PREFu to prefetch the address translation for the cache line and makes the prefetch succeed.

ARMv8 offers four flavors of prefetching instructions [39]. So far we have used a load keep prefetch which prefetches a line in shared state for a load. There is also a streaming variant for cases where the data does not need to stay in the cache for extended periods of time and may be replaced shortly after the access. Further, the ISA offers a store prefetch in keep and streaming flavor which prefetches a line in exclusive state. The ThunderX implements streaming or transient accesses by not setting the used bit in the cache line which makes it much more likely to be replaced. Experiment 3.4 shows that the load prefetches for both variants work as expected. Store prefetches, however, seem to be ignored and can also not be used to upgrade an L2 cache line in shared state to exclusive state.

***Following along:* Experiment 3.4: Prefetching Variants**

Required bitstream: None
Scirpt: `experiments/prefetch-variants.sh`
What you will see: Two tmux panes with a writing core on the left side and a reading core on the right will open. First, the writing core writes to five cache lines and writes them back to memory invalidates them. The reading core then tries to prefetch each cache line with a different variant of prefetch instruction.

The behavior of the ThunderX to ignore store prefetches for CPU-homed L2 cache lines makes sense when we remind ourselves that the L2 cache is shared. Therefore, if a CPU core wants to modify a cache line, this core will hold the cache line in exclusive state and the corresponding cache line in the L2 cache will be invalidated to maintain coherence. Thus, CPU-homed cache line in exclusive state does not really make sense. For remote cache lines however, a store prefetch should work as expected because the exclusive state in the L2 state signifies that the remote node has invalidated its copy of the cache line. This is indeed confirmed below by experiment 3.5.

Now that we know how to make the ThunderX reliably prefetch a CPU-homed cache line, we need to see if this also applies to FPGA-homed cache lines, which is what experiment 3.5 does. Due to the difficulties in meeting timing on a bitstream with the fully fledged DCS (cf. section 4.4.5), we use the single cache line copy from the dynamic stub (see section 2.2.2). This cache line controller copies the data from cache line `0x1800000080` to the next cache line `0x18000000100` when the former is modified by a write from the CPU. This causes the second cache line to be invalidated in the CPU cache.

Experiment 3.5 also shows that the ThunderX does not ignore store prefetches when the prefetched cache line is on a remote node. While we do not have concrete proof, a plausible explanation for this behavior might be that the ThunderXs write-buffers are more effective when prefetches are not interfering in their operation as a prefetch would flush the buffer in any case while a write to a different subcacheline might allow for more write-combining.

***Following along:* Experiment 3.5: Prefetching FPA-homed addresses**

Required bitstream: C3 no instruction test or shell stub
Script: `experiments/prefetch-remote.sh`
What you will see: The test will write two cache lines in the FPGA DRAM address space. A write to the first cache line causes the FPGA to copy the cache line to the second cache line, thereby invalidating the second cache line in the cache of the CPU. The test then demonstrates that prefetching that second cache line works. After writing another value, the reading core demonstrates, that exclusive prefetches are not ignored on remote addresses.

In experiment 3.5 we see that the FPGA returns the prefetched cache line in exclusive state even though we requested the shared state. Tests with the DCS showed, that it returns cache lines in shared state. This discrepancy heads from some freedom in the coherence protocol where directory controllers may return a cache line in exclusive state. However, this is completely unrelated to store prefetches not being ignored, because the ThunderX does not know in what state it will get a cache line when it decides whether to effectively issue a prefetch.

In summary, we have seen that to reliably prefetch a cache line using ARM's `prfm` instructions, we need to ensure that the address mapping for the access is cached in the TLB for the ThunderX does not ignore the prefetch. For the C3 this will require to add a PREFu CvmCache instruction before every prefetch.

Fetch and Lock

In addition to the ARMv8 `prfm`/`prfum` prefetching instructions the ThunderX offers the L2 Cache Fetch and Lock CvmCache system instruction. This fetch and lock instruction takes a physical address as an argument and fetches the cache line at this address into the L2 cache, performing cache replacement if nessecary [43]. Additionally, the fetch and lock instruction sets the used and lock bits in the specified cache line[3]. Crucially though, if all cache lines in all available ways (i.e. all ways the issuing core is allowed to access according to way partitioning) of the specified cache set are locked, then the fetch and lock instruction becomes a NOP. Experiment 3.6 demonstrates that the ThunderX executes a fetch and lock instruction even if the mapping for the affected cache line is not cached in any TLB. Cache lines which are locked need to be unlocked using an explicit invalidation, e.g. from an L2 Cache Hit Invalidate CvmCache instruction [43].

[3] If the specified cache line was already in the L2 cache, the fetch and lock instruction will still set the used and locked bits. In that case, it operates akin to a lock L2 cache line instruction

***Following along:* Experiment 3.6: Fetch & Lock homed addresses**

Required bitsream: None
Script: `experiments/fetch-lock-homed.sh`
What you will see: Two tmux panes with a writing core on the left side and a reading core on the right will open. First, the writing core writes to a cache line and writes it back to memory and invalidates the cache line. The reading core then uses L2 Fetch and Lock to fetch the prepared cache line in spite of the uncached address mapping. Then it will demonstrate unlocking the cache line using an L2 Hit Invalidate instruction.

The fetch and lock instruction is an attractive instruction for cache injection, because it operates on physical addresses and is thus not dependent on the paging or TLB state on the core. Further, locking a cache line will prevent the ThunderX from replacing it which ensures that the application is able to access the cache line as prefetched by the C3. But we need to be careful with locked cache lines as they may clog the L2 cache with cache lines that cannot be replaced which is bad for the entire system, but also for C3 if it cannot inject any more cache lines.

In experiment 3.7 we use the FPGA with its cache line copy mechanism to demonstrate that fetch and lock works for FPGA-homed addresses and that a locked cache line is unlocked by the invalidation resulting from a write to the same cache line from another node. We do this by writing to the first cache a second time which causes a write to the second cache line on the FPGA, which in turn invalidates and unlockes the locked cache line in the L2 cache on the CPU.

***Following along:* Experiment 3.7: Fetch & Lock remote addresses**

Required bitstream: C3 no instruction test or shell stub
Scirpt: `experiments/fetch-lock-remote.sh`
What you will see: The test will write two cache lines in the FPGA DRAM address space. A write to the first cache line causes the FPGA to copy the cache line to the second cache line, thereby invalidating the second cache line in the cache of the CPU. The test demonstrates that the L2 Fetch and Lock instruction successfully fetches a remote cache line into the CPU L2 cache and, by writing to the first cache line again, that locked cache lines can be unlocked by the invalidation caused by the write on another cache line.

If an application on the FPGA issues direct cache injection C3 instructions and is subsequently turned off or going idle, the union of cache lines injected into the L2 cache over its runtime will remain locked and can thus never be replaced unless invalidated explicitly. Thus, if a C3-aware application uses the fetch and lock instruction and wants to support graceful shutdown, it must take care to invalidate all locked cache lines during its shutdown procedure. Therefore, the fetch and lock instruction should also only be

utilized for prefetching CPU-homed cache lines in more involved prefetching patterns, if these cache lines are invalidated at one point.

Performance Comparison

Since we are interested in low latency data transfers, we measure the latency of a cache injection using a load, a prefetch enabled by a PREFu, and a fetch and lock. Because these instructions exhibit different behavior based on the state of the TLBs, we perform a measurement where the cache injection happens after a TLB flush.

To test the cache injection, we use a bitstream with the cache line copy on the FPGA, write a fresh value to cache line l_1, the FPGA copies the data written to l_1 over to the next cache line l_2. Then we time the cache injection of l_2 into the CPU L2 cache using one of our three instructions. For the test cases with a TLB flush, we add a `tlbi all` instruction before the timed cache injection. We run this procedure in a loop for 10'000 iterations and record the measurements. We can see the result of experiment 3.8 in figure 3.1.

Since we are measuring the latency of one or two instructions, we must be careful to place appropriate barriers to measure the whole effect of the fetch. As a counter we use the `cntvct_el0` system counter running at a fixed frequency, also readable in the `cntfrq_el0`. We use an instruction synchronization barrier (`isb`) and a data synchronization barrier (`dsb sy`) before reading the counter and an instruction barrier after reading the counter as suggested by the ARM ARM [39]. This is to ensure that all the counter read cannot be reordered and that all loads and stores complete before the counter is read. Additionally, we have to place barriers to ensure that the previous iteration of the loop has completed and its effect are visible, and that l_2 is invalidated before the instruction under test fetches it.

Following along: **Experiment 3.8: L2 Cache Fetch Latency**

Required bitstream: C3 no instruction test or shell stub
Script: `experiments/fetch-latency.sh`
What you will see: First, the measurements as described above are run on the isolated core 47 to avoid interference from other processes and reduce OS jitter. Then, the results get plotted and written to `data/fetch-latency.{csv,png}`.

Figure 3.1 shows the violin plot of the measurements from experiment 3.8. The plot shows the latency in nanoseconds on the y-axis and the respective instruction that performed the fetch on the x-axis. For each instruction we have the results for the measurements without TLB invalidation on the left and with TLB invalidation on the right. Experiment 3.8 was run on `zuestoll08` with the C3 no instruction test bitstream loaded on the FPGA. The program `usr/fetch-latency.c` was compiled using `gcc 9.4.0-1ubuntu1~20.04.1`. All

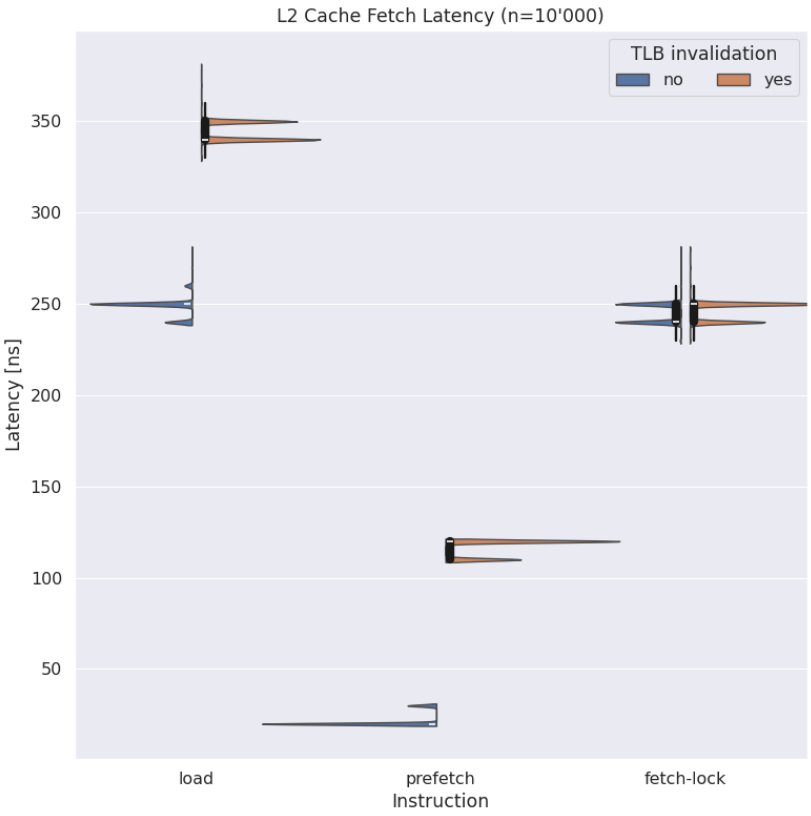

Figure 3.1: Latency to fetch one cache line from the FPGA into the CPU L2 cache.

CvmCache instructions were executed from EL0 as the experiment sets the required bit in the `CVM_ACCESS_EL1` register.

Immediately, we can see that the measured latency for the PREFu/prefetch instructions in the middle is much lower with a mean of 21.42 ns without TLB invalidation and 116.98 ns with TLB invalidation is much lower that for the other two instructions. This is well below the interconnect latency for two connected ThunderXs of roughly 150 ns [10] and thus this is certainly not the latency between issuing PREFu and the prefetched data arriving in the L2 cache. However, this is contrary to the ThunderX manual which states that a `dsb sy` will wait for all prior load and prefetch operations to complete. Thus, either prefetching using `prfm` instructions does not work in experiment 3.8 or the completion of a prefetch operation on the ThunderX is different from the data arriving in the cache.

We can see further, that only the fetch and lock instruction, which operates on physical addresses, is not affected by the TLB flush. The load instruction and the PREFu/prefetch

instructions are roughly 100 ns slower when the TLB has been flushed and suggests that a page table walk has taken place. The load instruction without TLB invalidation and the fetch and lock instruction have roughly the same latency at with a mean of 248.66 ns and 244.95 ns respectively.

Comparing the distributions of all sets of measurements in figure 3.1 we can see that the distributions spike about every 10 ns. This is due to the resolution of the system timer `cntvct_el0` which runs at a frequency of 100 MHz on Enzian.

Utility

After investigating the behavior of three different methods for achieving L2 cache injection on the ThunderX and measuring their performance, we draw some conclusions on their utility in the C3.

The main goal of direct L2 cache injection from the FPGA is to transfer a cache line from the FPGA into the CPU L2 cache such that an application on the CPU can access that cache line without incurring a miss in the L2 cache. One way of reading this goal is that we want as low latency as possible.

This reading would suggest to always use a prefetch instruction preceded by a PREFu to fetch a cache line into the L2 cache as this allows the line to be prefetched while the C3 can already go off and fetch the next C3 instruction. However, when the L2 cache is highly contested, the fetched cache line might already have been replaced by a competing workload before the application on the CPU has managed to access it. A load suffers from the same problem, albeit with roughly 200 ns higher latency. Transient prefetches will, due to their inherent nature, fare even worse and therefore be of limited use to C3.

To guarantee that a fetched cache line remains in the L2 cache until the application accesses it, we must lock the cache line by fetching it using the L2 Cache Fetch and Lock CvmCache instruction. While it is 100 ns to 200 ns slower, the CPU application will never incur a miss in the L2 cache on injected cache lines. The fetch and lock instruction has the requirement that it will only replace unlocked cache lines in the L2 cache. Due to the FPGA application having to write the data to memory first, the coherence protocol will invalidate and thus unlock all cache lines the C3 will have to fetch.

In summary, due to its guarantees of fetched cache lines remaining in the cache, the fetch and lock instruction seems most promising for use with the C3. However, instead of locking a cache line, we could also prevent aggressive workloads from evicting fetched cache lines using cache allocation.

3.3.2 Cache Allocation

Cache allocation refers to mechanisms which reserve parts of a shared cache to a subset of workloads using the shared cache. For example, the L2 Cache Fetch and Lock CvmCache instruction is a form of cache allocation as it reserves a locked cache line to workloads accessing this specific cache line. The ARMv8 architecture features cache allocation hints complementing the cacheability attributes, which implementations can use to "limit cache pollution to a part of the cache" [39]. Unfortunately, the ThunderX does not make use of cache allocation hints [43].

But the ThunderX L2 cache has a feature called way partitioning. Using bitmasks in system registers software can restrict access to certain L2 cache ways for each core and each I/O bridge [43]. It is important to understand that way allocation on the ThunderX only affects replacement. Hence, any core can read and invalidate (for exclusive access) any cache line in the L2 cache. However, only cores which are inside a way partition are allowed to replace cache lines. This cache allocation feature allows us to isolate one or multiple L2 cache ways for the C3 and cores running the application from conflict misses originating from other cores However, removing one 16^{th} of the total L2 cache capacity for the rest of the system may have a significant impact on the overall performance of the machine as a whole. The tradeoff between removing contention in push-prefetching as observed by Farshin et al. [22] and overall system performance needs to be reevaluated for each C3-aware application depending on how many cache lines it wants to inject into the L2 cache at a time.

Similar to the fetch and lock instruction, the applications need to take care to deallocate allocated ways on shutdown or even dynamically when they are idle[4]. In fact, this is more important than with locked cache lines as at least 1 MiB will not be accessible to the rest of the system. Further, we must take care that the C3 is always allowed to access the allocated L2 cache ways as it would not fetch cache lines into the allocated way otherwise, which would entirely defeat the purpose. Unfortunately, for the C3 to fetch all cache lines of an application into the allocated way is only possible if the allocated way is the only way the C3 can access in the L2 cache. This is because the ThunderX places cache lines in the first available way in the L2 cache. However, we cannot control in which way a cache line is placed except if there is only one way the writing core is allowed to replace blocks in by use of way partitioning. However, it is unclear if reserving a single way for the C3 is more beneficial than restricting the access of other memory heavy applications on the system to a subset of the ways. Unfortunately, due to time constraints we were not able to conduct any tests regarding way partitioning.

[4]It is not clear at this time whether repeated allocation and deallocation is more beneficial than having a way reserved constantly if the system is under heavy load. Intuitively, giving back more of the cache to a memory intensive application should be beneficial, but taking it away unexpectedly might also lead to unforeseen access patterns and thrashing.

With this, arguably quite limited, cache allocation capability available we need to control it effectively. In principle both the CPU application and its counterpart on the FPGA can control the way partitioning. If the applications need a way allocated for the entire runtime, it is easier to have the CPU side control way partitioning as it can be done during initialization and cleanup of the CPU application and does not need a round trip via the FPGA. If, however, the FPGA knows that based on a particular request from the CPU or perhaps network traffic that a lot of data needs to be transferred in the foreseeable future, it would be beneficial to have it dynamically allocate a cache way using a C3 instruction. In general, applications must use way partitioning thoughtfully and benchmark different configurations to ensure that the performance of other workloads is not drastically diminished. Further, applications must take care to not starve any cores or I/O-bridges of replacing cache lines in the L2 cache.

3.3.3 Core Isolation

Above in section 3.2.1 we decided that the Cache Control Coprocessor should be implemented as an isolated core that continuously polls for new C3 instructions. In this section we investigate different methods of isolating a core on Enzian and decide on a method to use for the C3 implementation for this work.

The first method for core isolation is to run the C3 as kernel thread on a core isolated using the `isolcpus` command line parameter for Linux. By writing an out of tree Linux kernel module we are afforded with (almost) all the capabilities and amenities of the Linux kernel. We trivially have access to all memory in the system and all system registers and instructions only available from EL1. Further, we can use the file abstractions to implement notifications. A no data notification can be implemented using the poll system call to wait for a new notification and a notification with data can be implemented using the read system call. However, using these facilities incurs the penalty of a context switch on notification.

Because the Linux kernel is monolithic by design it is hard to isolate a core fully. In terms of isolation, the `isolcpus` command line parameter gets us most of the way there. The `isolcpus` parameter provides three options to isolate a list of CPUs from [12]. The `nohz` flag disables the timer interrupt or tick on the listed CPUs. The `domain` flag isolates the CPUs from simultaneous multiprocessing (SMP) balancing and scheduling algorithms by putting them in a special scheduling domain. This domain isolation cannot be changed during the runtime of the system. The `managed_irq` flag isolates the specified CPUs from managed interrupts, by removing the isolated CPUs from the respective interrupt request (IRQ) affinity list. This leaves still some management kernel threads on an isolated CPU. Most importantly, we have to make sure that the Read, Copy, Update (RCU) subsystem does not starve when we run the C3 fetch loop as it is central to keep the Linux kernel functioning. To isolate CPUs from RCU callbacks, we can use the `rcu_nocbs` kernel command line parameter which offloads the callbacks to dedicated kernel threads on other

cores. However, the offloaded cores are required to periodically wake up the offloaded kernel threads. To relieve this burden from the isolated core we use the `rcu_nocb_poll` command line parameter which makes the offloaded kernel threads wake up periodically without any prompting form an isolated CPU.

The second option guaranteeing a fully isolated CPU core is the work of Walters master thesis [61] which uses Linux as a boot loader. This was achieved by effectively shutting down a CPU, but replacing the PSCI `cpu_off()` function with a custom function that jumps to the entry point of a bare metal program. This bare metal program then runs on a core that from Linux's point of view is off and thus fully isolated. Before calling the isolated program, the Linux kernel sets up page tables in a memory region reserved at boot so the bare metal program does not have to install pages itself and does not conflict with the Linux kernel.

On the isolated core we would run in EL1 so we have the same privileges as in the first variant, but without the comforts of the abstractions that Linux provides. Although, this also affords us some freedoms. One concern with an isolated kernel thread is that it is very hard to map huge pages for use in the kernel. It would be useful to map the entire FPGA DRAM address space using 16 GiB huge pages to increase the address space mappings in the TLB cover. Running bare-metal, we are free to install these ourselves — but we have to do everything ourselves.

By default, we also do not have any communication from the isolated core to the rest of the system. In the thesis, this was solved by implementing a virtio interface on the isolated core and a kernel module on Linux implementing a virtio driver. This way the isolated was able to implement a serial device and provide output to the outside world. Since the virtio interface can abstract many kinds of I/O, the C3 should also be able to use it for notifications. Further, cross core communication using some UMP like mechanism would also be possible.

The third option for isolating the C3 is to run it as a userspace process and isolate a core the same way as in the first variant at boot time or using cgroups at runtime. While we can issue all CvmCache instructions from EL0 with the correct setting in `CVM_ACCESS_EL1`, we would still need some function implemented in the kernel to issue SGIs for example. However, from userspace it is also easier to map the entire FPGA DRAM address space using huge pages. Cross core communication between the C3 and an application would need to use some UMP mechanism or some Linux interprocess communication (IPC) mechanism like Unix domain sockets for a lower level of abstraction or D-Bus for a higher level of abstraction. All in all it seems unlikely for a userspace C3 to have as low latency as the first two options described above.

In light of the goals of this thesis, the first option of isolating a Linux kernel thread seems most promising as it provides more flexibility and extensibility with its facilities than a bare metal program where we would have to implement everything ourselves. While a

reimplementation of a Linux based C3 to a bare metal C3 might prove valuable after some time due to measurable deficiencies it is much more important for the scope of this work to have a solid foundation and a basis for experimentation.

Design Decision 9. The C3 is implemented as a Linux kernel module and the Linux facilities are used to isolate it on a core.

3.4 C3 Instructions

At the heart of the C3 system is the fetching and execution of C3 instructions. Since we fetch instruction from the FPGA I/O address space our address size is 64 bits. First, we describe the encoding in these 64 bits and then we describe the different C3 instructions in detail.

3.4.1 Encoding

Instructions need opcodes. To be useful in all cases, a C3 instruction should be able to carry an opcode and a full 57-bit ARMv8 tagged virtual address. This leaves us with a 7-bit opcode. We restrict the instruction encoding to four different *forms*. The forms represent a common structure for encoding and decoding to be easier. To further simplify decoding, we use the most significant two bits of the opcode to encode the form of our instruction (see 3.2). Hence, we have space for 32 *functions* or different instructions per form. With only three forms needed at the time of writing, this encoding can be extended with one more form or alternatively one form can be expanded to have 64 instructions. In principle, a specific function could specify the effect of an instruction and the form only specifies the encoding akin to different addressing modes in other ISAs. For C3 we do not go this route, as this would leave us with only 32 instructions and a capability which is not needed as such. Functions in the context of opcodes are thus merely a name for the least significant 5 bits of a C3 opcode.

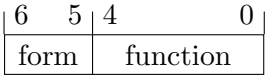

Figure 3.2: Encoding of the opcode of a C3 instruction.

For even easier decoding, we also give the opcode a fixed place across all forms, so a simple bitmask can read the form and function of the instruction. For direct cache injection of cache lines, C3 instructions must carry the addresses of cache lines. Since the cache line size on Enzian is 128 B, the least significant 7 bits of that address are irrelevant for addressing the cache line. Therefore, we put the opcode in the least significant 7 bits for all forms of C3 instructions.

Figure 3.3: The form 0 encoding of a C3 instruction.

Form 0 When preforming direct cache injection of multiple consecutive cache lines, the FPGA application should not have to issue multiple C3 instruction. From this requirement we draw the need for ranged C3 instruction to enable the direct injection of a range of cache lines. For a cache line injection we obviously also need the address (physical, of course, as discussed in section 3.2.1) of the first cache line in the range. To specify any physical address in the DRAM address space on the ThunderX we need 42 bits [43], although on Enzian we only need 41 bits because we only have two nodes. However, the extra bit simplifies the encoding overall, as it makes the field boundaries of form 0 and form 1 line up. As mentioned above, we do not need the seven least significant bits of this address, because we are addressing a 128 byte cache line. We use 13 bits for the length field, so we can transfer 1 MiB or one way of the L2 cache with one instruction as decided in design decision 1. To be able to transfer the full1 MiB, the length field actually encodes the length minus one. Hence, to transfer one cache line, we set the length field to zero. This leaves 9 bits we can use for additional data like an application ID. Figure 3.3 shows the encoding of form 0.

Decoding form 0 is very efficient as the opcode and the cache line address can be obtained with a single bitwise and operation and do not require any shifting. These two values are also the first values needed when executing a cache injection instruction. The length field is also not needed until after the first cache line fetch, so its slightly less efficient decoding could be pipelined with the fetch operation.

Form 1 The form 0 of a C3 instruction is useful for addressing cache lines, but sometimes we might also need to address a full 48 bit physical address. This is exactly the functionality form 1 provides. As shown in figure 3.4, it features the opcode and additional data in the same location as form 0, but, in between those two fields, the 48 spare bits neatly fit the full physical address.

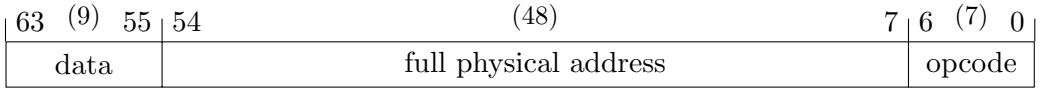

Figure 3.4: The form 1 encoding of a C3 instruction.

Form 2 To enable the biggest argument possible in a C3 instruction, we have form 2. It features only the opcode in the 7 least significant bits and a 57-bit field for data, which is large enough for a full 57-bit ARMv8 tagged virtual address. Form 2 is also the way

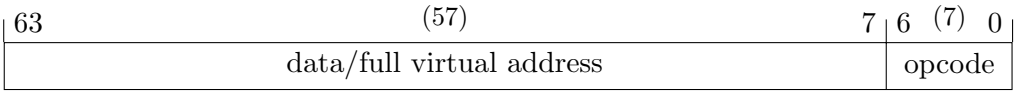

Figure 3.5: The form 2 encoding of a C3 instruction.

to go if an instruction has no associated data to transfer. Figure 3.5 shows the simple encoding of form 2.

3.4.2 Instruction Descriptions

In the following subsections we find the descriptions of the concrete C3 instructions. For each C3 instruction its opcode with form and function, its operation and design considerations are given. We can group the instructions into the following groups: control instructions, prefetch instructions, cache management instructions, and notification instructions.

Control Instructions

Control instructions provide error signal capabilities to FPGA applications and the C3 envoy. At the time of writing, the semantics of error handling on the FPGA have not been investigated intensively. Thus, the design of the instructions below is mostly uninformed and should be understood as a starting point for further investigation.

No Instruction opcode `0x00` (form 0, function `0x00`)
This is the NOP of the C3 instructions. It is the `NULL` instruction which is ignored by the C3 and continues the fetch loop in the next iteration. The only reason that it has form 0 is that the opcode is all zeroes.

Error opcode `0x2e` (form 1, function `0x0e`)
The error C3 instruction signals a recoverable error condition. The data field contains an application defined error number. The C3 logs errors and continues its fetch loop.

Fatal Error opcode `0x2f` (form 1, function `0x0f`)
This instruction signals a fatal error to the C3 which logs it as such and stops its operation because it has to assume that the C3 envoy has encountered the error.

Prefetch Instructions

The main purpose of the prefetch instructions is direct cache injection. However, they can also be used to prefetch data on the CPU the FPGA application knows will be needed based on its knowledge. All prefetch instructions are form 0 and use the address and length fields as designed, but the data field is unused.

Fetch and Lock opcode `0x10` (form 0, function `0x10`)
The Fetch and Lock C3 instruction prefetches the specified cache lines into the CPU L2 cache using the L2 Cache Fetch and Lock CvmCache instruction. All caveats from section 3.3.1 apply.

Prefetch Shared opcode `0x11` (form 0, function `0x11`)
The Prefetch Shared C3 instruction uses a `prfm pldl2keep` instruction preceded by a PREFu CvmCache to fetch the specified range of cache lines into the CPU L2 cache. All caveats from section 3.3.1 apply.

Prefetch Shared Streaming opcode `0x12` (form 0, function `0x12`)
The Prefetch Shared Streaming C3 instruction uses a `prfm pldl2strm` instruction preceded by a PREFu CvmCache fetch the specified range of cache lines into the CPU L2 cache. All caveats from section 3.3.1 apply. In particular, cache lines prefetched with this instruction tend to be replaced shortly after they have been fetched.

Prefetch Exclusive opcode `0x13` (form 0, function `0x12`)
The Prefetch Exclusive C3 instruction uses a `prfm pstl2keep` instruction preceded by a PREFu CvmCache to enable the prefetch to succeed. All caveats from section 3.3.1 apply.

Prefetch Exclusive Streaming opcode `0x14` (form 0, function `0x14`)
The Prefetch Exclusive Streaming C3 instruction uses a `prfm pstl2strm` instruction preceded by a PREFu CvmCache to fetch the specified range of cache lines into the CPU L2 cache. All caveats from section 3.3.1 apply. In particular, cache lines prefetched with this instruction tend to be replaced shortly after they have been fetched.

Load opcode `0x15` (form 0, function `0x15`)
The Load C3 instruction uses a load to access a cache line and thus also fetch these cache lines into the CPU L2 cache. Note that this instruction pollutes the L1d cache of the

C3 and thus potentially causes the C3 loop to incur conflict misses which causes higher latency between instruction fetches.

Cache Management Instructions

The cache management instructions expose features to FPGA applications usually reserved to CPU applications for managing the L2 cache. Depending on the bitstream on the FPGA, the DCS can perform some of these operations for FPGA-homed cache lines, but not for remote cache lines from the view of the DCS.

All cache management instructions except those concerned with cache allocation use form 0 the same way as prefetch C3 instructions since they also operate on ranges of cache lines.

L2 Zero opcode `0x01` (form 0, function `0x01`)
The L2 Zero C3 instruction uses `dc zva` to zero a range of cache lines in the CPU cache. The C3 must ensure that the system register `DCZID_EL0` is set such that `dc zva` actually operates on the granularity of a cache line. It may set the register to larger values to zero multiple cache lines with a single instruction, but then it would need to reset the value right after that call. Since this instruction behaves like a store it may pollute the L1d cache of the C3.

L2 Invalidate opcode `0x02` (form 0, function `0x02`)
The L2 Invalidate C3 instruction uses the L2 Cache Hit Invalidate CvmCache instruction to invalidate a cache line at a specific physical address. This instruction must be used with care as it causes data loss if it invalidates a cache line with dirty data that has not been written back at the time of invalidation.

L2 Writeback Invalidate opcode `0x03` (form 0, function `0x03`)
The L2 Writeback Invalidate C3 instruction uses the L2 Cache Hit Writeback Invalidate CvmCache instruction to first write back and then invalidate a cache line at a specified physical address. This operation can be performed by the DCS for FPGA-homed cache lines with a local-clean-invalidate transaction [51].

L2 Writeback opcode `0x04` (form 0, function `0x04`)
The L2 Writeback C3 instruction uses the L2 Cache Hit Writeback CvmCache instruction to write back an L2 cache line at a specified physical address. This operation can also be performed by the DCS for FPGA-homed cache lines with a local-clean transaction [51].

L2 Partition Way for Cores opcode `0x20` (form 1, function `0x00`)
The L2 Partition Way for Cores form 1 C3 instruction partitions a specified way of the CPU L2 cache to the cores specified in the CPU mask. While all cores are still able to access all L2 cache lines, only the specified cores will be able to replace cache lines in the specified way. Note that this instruction does not affect the way partitioning of IOBs. Use this instruction in conjunction with L2 Partition Way for IOBs to fully configure the partition for a way. See figure 3.6 for the encoding of the instruction.

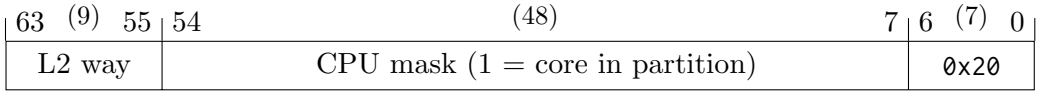

Figure 3.6: Encoding of the L2 Partition Way for Cores C3 instruction.

L2 Partition Way for IOBs opcode `0x21` (form 1, function `0x01`)
The L2 Partition Way for IOBs form 1 C3 instruction partitions a specified way of the CPU L2 cache to the IOBs specified in the IOB mask. While all IOBs are still able to access all L2 cache lines, only the specified IOBs will be able to replace cache lines in the specified way. Note that this instruction does not affect the way partitioning of CPU cores. Use this instruction in conjunction with L2 Partition Way for Cores to fully configure the partition for a way. See figure 3.7 for the encoding of the instruction.

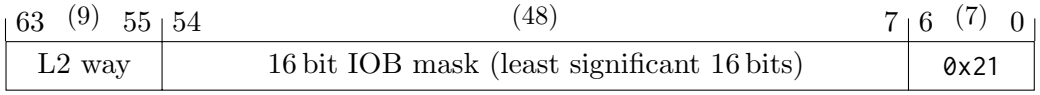

Figure 3.7: Encoding of the L2 Partition Way for IOBs C3 instruction.

Notification Instructions

Notification instructions offer an FPGA application a variety of ways to let the CPU know about an event. The abstract notification instructions notify an application on the CPU using the method the application registered with the C3 based on the specified ID. The SGI instructions cause the C3 to issue an SGI on the CPU.

Notify No Data opcode `0x30` (form 1, function `0x10`)
The Notify No Data C3 instruction causes the C3 to notify the application specified with the application ID that a notification has arrived. This instruction is similar in nature to an interrupt. See figure 3.8 for the encoding of this instruction and note that this instruction has form 1 for reasons of consistency with the other notification instructions which also have the application ID in the data field in the nine most significant bits of the instruction.

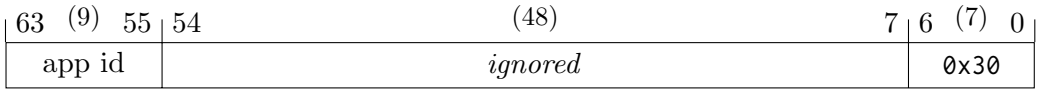

Figure 3.8: Encoding of the Notify No Data C3 instruction.

Notify Cache Line Range opcode `0x0a` (form 0, function `0x0a`)
The Notify Cache Line Range C3 instruction is used to notify the CPU application specified in the data field with its application ID that data is ready in a range of cache lines. The cache line range is associated data which is included in the notification to the CPU application.

Notify Data opcode `0x31` (form 1, function `0x11`)
The Notify Data C3 instruction is used to notify a CPU application specified by its application ID with associated data. This data is included in the notification to the application. See 3.9 for the encoding of this instruction.

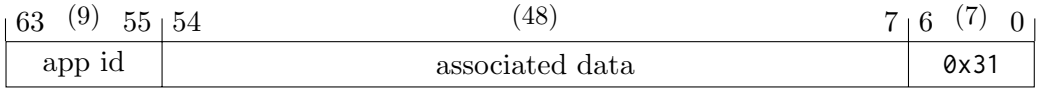

Figure 3.9: Encoding of the Notify Data C3 instruction.

SGI 8 – 15 All Cores opcode `0x48-0x4f` (form 2, function `0x08-0x0f`)
The SGI All Cores family of C3 instructions issue an SGI to all cores on the CPU. The GIC interrupt ID is specified in the function of the opcode. Data passed in the 57 bit data field is copied into a list of SGI vectors maintained by the C3. See figure 3.10 for the encoding of this instruction. Applications need to coordinate the eight interrupt lines among themselves. Ideally, the FPGA application can be configured to use a specific interrupt number by its CPU counterpart depending on which interrupt it was able to acquire.

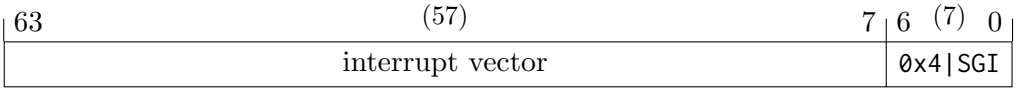

Figure 3.10: Encoding of the SGI All Cores family of C3 instructions. `SGI` is the 4-bit interrupt ID of the SGI.

SGI 8 – 15 Core Mask opcode `0x38-0x3f` (form 1, function `0x18-0x1f`)
The SGI Core Mask family of C3 instructions issue a SGI to the cores specified in the core mask. The GIC interrupt ID is specified in the least significant nibble of the opcode (all bits except the most significant bit of the function). The 9 bit data field is copied

into the list of SGI vectors maintained by the C3 before the cores are interrupted. See figure 3.11 for the encoding of this instruction. Applications need to coordinate the eight interrupt lines among themselves. Ideally, the FPGA application can be configured to use a specific interrupt number by its CPU counterpart depending on which interrupt it was able to acquire.

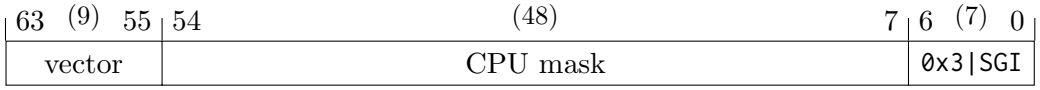

Figure 3.11: Encoding of the SGI Core Mask family of C3 instructions. `SGI` is the 4-bit interrupt ID of the SGI.

3.5 Application Interface

After having designed the internals of the C3 system, we look at the interface to applications in this section. On the FPGA the only interface to C3 is the ready/valid interface to submit C3 instructions into the instruction queue of the C3 envoy. This interface can also easily be multiplexed for multiple C3-aware applications on the FPGA. On the CPU though, there are two principle interfaces of the C3 to applications: application registration and notifications from C3 instructions. Knowing that the C3 is implemented as a Linux kernel module (cf. design decision 9), we can design those more concretely.

3.5.1 Application Registration

With the C3 implemented as a Linux kernel module, we have applications register themselves with the C3 using an ioctl. With this registration ioctl, applications provide their application ID and a notification method. Using this, the C3 can match the application ID from notification instructions to IDs registered with it from applications and send notifications using the preferred method. To allow the applications to choose their ID themselves, the C3 provides a large enough ID space to make ID collisions unlikely.

An application can, in principle, register itself multiple times with the C3, for example to be able to distinguish between two kinds of notification or notifications from two different FPGA applications. However, if it does so, it needs to open the C3 file again for every new registration. This allows the C3 to track application information on a per file basis.

The C3 can limit the number of concurrent registered applications to maintain low-latency guarantees. It also provides an ioctl for applications to unregister. New applications can reuse application IDs used previously, as long as they do not collide with any IDs registered at the time of registration.

3.5.2 Notifications

Notification instructions leave the C3 with the task of delivering these notifications, often with associated data to the application on a different core. Because the C3 is a kernel module, can use all the Linux kernel facilities for cross-core communication. Further, we can use file operations like read of poll to notify the application.

As mentioned above, C3-aware applications register themselves with the C3 and provide a notification method. With a notification method, the application tells the C3 at least which interface and what guarantees it expects. The real design decision is whether the notification message also specifies the data format the application expects. If the data format, i.e. no data, cache line range, and data, is fixed for each method, then we need the cross product of available interfaces and data formats as notification method. Also, not ever notification instruction can send a notification to every registered application ID, because the data format might not fit. A more flexible approach would be to have a data format that tells the application how to interpret the data, because then all notify instructions could send notifications to all registered applications.

Design Decision 10. All notifications deliver data in the same format that contains a format-tag. The tag serves applications as the information on how the delivered data should be decoded.

Each registered notification needs to keep some state to store the delivery method and data structures needed for notification delivery. For some delivery methods, this notification state needs to be shared across cores. File operations like `read(2)` run on the same core as the application that invoked the system call. Thus, this data is shared memory for many notification mechanisms and needs to be protected accordingly. The mechanism for protecting this shared memory can be different for every notification method. However, we want to avoid cache line thrashing between the caches of the C3 and the application core. Thus, methods with which the application core only has to read the shared data are preferred. Further, the data for each notification resides on its own cache line to avoid thrashing cache lines between applications.

Notifications over File Abstractions

With the C3 implemented as a kernel module, it can implement file operations to deliver notifications to applications. First, an application needs to `open(2)` the device file of the C3 and use it to register itself using the `ioctl(2)` described above. Once registered, the C3 will deliver notifications over `poll(2)` and `read(2)`. The operation of the two depends on whether the application has opened the file in a non-blocking way, i.e. with the flag `O_NONBLOCK`.

By poll(2)ing the C3 in a non-blocking fashion, an application can find out if a notification has arrived yet and do other work while waiting. In non-blocking mode, poll(2) returns immediately, only giving information if a notification has arrived. In blocking mode, poll(2) blocks the execution of the application until a notification arrives. Usually, poll(2) is used to wait on data in a file descriptor to become ready to read (or write) which it will signal with a POLLIN flag as a return value. But because we have notifications which are only events without associated data, it would be advantageous to be able to distinguish this from cases where the application will still need to read data. The flag POLLPRI is used to signal "some exceptional condition on the file descriptor". We use it in C3 to signal that the received notification does not have any associated data if it is possible to do so.

By read(2)ing the C3 file descriptor in a non-blocking fashion, we can achieve about the same functionality as with poll(2). When no notification has arrived yet, read(2) will then return EWOULDBLOCK to tell the application to try again later. In blocking mode, read(2) will block until a notification arrives. Once a notification has arrived, read(2) copies the requested amount of data into the application provided buffer. In this case, it is the application's job to determine if this is a no-data notification based on the format tag in the data it received[5].

With the file abstractions, notification delivery can be both synchronous and asynchronous depending on whether the application has opened the file with the non-blocking flag. Therefore, we must assume asynchrony and buffer notifications until they are delivered. However, with asynchronous notification delivery we can run into the problem where the next notification arrives before the current notification has been delivered to the application. There are essentially three ways to deal with this: buffer multiple notification in a queue, drop the old notification, or drop the new notification. Because all three have valid use cases, C3 offers all of them as variants for notification delivery.

UMP-like Notification Delivery

Using the file abstractions as our notification method has the drawback that we pay the cost of a context switch on top of a cross-core communication. Ideally, we would only have to pay for the cross-core communication without paying for a context switch. Using the idea of Barrelfish's UMP [3], which in turn is based on the ideas of URPC [5] to avoid context switches and FastForward [26] as a technique, we use shared memory to communicate synchronously between the C3 and the application.

The core idea of UMP-like Notification Delivery (UND) is to use a cache line shared between the C3 and the application for notification delivery using the UMP transfer mechanism. This allows the application, polling the cache line from userspace, to receive

[5]Checking data to see if "no data" arrived is counterintuitive, granted, but a direct consequence of design decision 10

data written to the cache line by the C3 in kernel space without a context switch in the middle.

The data transfer mechanism of UMP leverages the cache coherence protocol to transfer a shared cache line between a writer and a reader polling the cache line. All coordination and data transfer is completed on the shared cache line which is parsed as a message. The two sides coordinate using a state field in this message.

First, we go over the sender's steps to send a message.

1. Check that the state of the previous message on the cache line is received (or the buffer was just initialized).
2. Write the message to the shared cache line.
3. Set the message state to sent.
4. Poll the shared cache line until the receiver sets the message state to received.

Note, that step 4 is not strictly necessary, but if we wanted the sender to synchronize with the receiver, this step would achieve that. This would also ensure that the first step is a simple check and does not need polling.

Second, consider the reader's steps to receive a message.

1. Poll the shared cache line until the message state is sent.
2. Copy the message from the cache line.
3. Set the message state to received.

In its first step, the reader will poll on the shared cache in its local L1d cache with the cache line in shared state until the steps 2 and 3 of the writer will cause the coherence protocol to invalidate the readers copy of the cache line. Due to always hitting in the L1 cache, the reader's polling is very efficient. The first time the reader polls the now locally invalidated cache line, the coherence protocol will fetch a local copy of the shared copy back into the readers L1d cache. At this point, the reader reaches step 2 and subsequently step 3.

In Barrelfish, UMP is implemented on a ring buffer of shared cache lines, so the reader could receive messages asynchronously to the writer sending them. However, UND for C3 initially requires the C3 and the application to synchronize on the notification. This is mainly for simplicity, so the method can be evaluated and improved before implementing the more complicated version. This synchrony, however, results in the UND notification method requiring that all applications using it need to synchronize between the CPU and FPGA such that the CPU side is always ready to receive a notification when the FPGA side sends a notification.

The crux with UMP is the setup of the shared memory on two different cores in different address spaces. UND faces the same challenge with the addition that the sender executes in kernel space and the receiver in userspace.

To set up a shared buffer with a kernel thread, we could have the userspace process allocate a buffer and pass a pointer to the buffer to the kernel. The kernel thread can then access the buffer using the Linux kernel API `get_user()`/`copy_from_user()` and `put_user()`/`copy_to_user()`. However, these functions involve address translations, but if the kernel thread does not perform polling that should be fine. However, these functions would not work in the C3 because they rely on the `ldtr` instruction on ARMv8, which performs a load as if the context was EL0. Because the C3 is a different kernel thread on a different core than the kernel thread which registered the application this will not load the desired location. But since the kernel knows the address mapping, it can calculate the physical address of the buffer allocated by the userspace process and write directly to that physical address using the Linux kernel's identity mapping of physical memory. While this would work, it violates some of the most basic assumptions and security principles in the Linux kernel or for operating systems in general.

A more principled, but also more complicated way is to have the kernel thread allocate a buffer. The userspace process can then `mmap(2)` the buffer into its address space. Now the buffer is not some memory pointed to by an untrusted user pointer, but memory managed by the kernel which is mapped properly. Using this extra call to `mmap(2)` we can get a shared buffer to use for UND.

SGIs as Notifications for Kernel Applications

So far, whenever we talked about a C3-aware application we implicitly talked about a userspace process on the CPU with an offloading workload on the FPGA that use the C3 for transferring data. However, workloads in the kernel can also be a C3-aware applications. One possible scenario would be a smart-NIC running on the FPGA which uses C3 to send incoming packets to the kernel networking stack using direct cache injection.

While communication between an application kernel thread and the C3 could in principle be established, we do not need any additional mechanisms to enable kernel applications. We can simply use SGIs as notification instructions. Kernel applications can install an interrupt handler for a given SGI, which also prevents two kernel applications from using the same SGI, and read from the SGI vector inside the handler to get the notification data. Since there are only eight SGIs available, the FPGA application will have to be told by the kernel application for which SGI it was able to install a handler.

3.6 Synchronization

This section goes into detail on the synchronization between the applications on the FPGA and the CPU with respect to C3 instructions. Basis for this discussion is design decision 4.

3.6.1 Provided Guarantees

Design decision 4 states that "the only explicit synchronization in the C3 system is the serialized execution of C3 instructions and notification instructions for FPGA applications to notify CPU applications". What guarantees does this design decision provide for the C3 system?

The serialized execution of C3 instructions guarantees that a C3 instruction i_1 is issued, i.e. it is written to the instruction queue of the C3 envoy, before another instruction i_2 *if and only if* i_1 has taken effect before i_2 starts executing on the C3. Taking effect means that the instruction has executed and effected the system, e.g. fetched memory into the CPU L2 cache. Note that for notification instructions this only guarantees that the notification was sent to the application and not that it was effectively delivered.

With this, notification instructions guarantee the CPU application that the FPGA application has reached the state signaled in the notification. But the FPGA application gains no knowledge whatsoever about the execution of the CPU application from the notification instruction. The C3 system only provides one way communication from the FPGA to the CPU. This missing feedback on the progress of the CPU application's execution is an important part of what the "synchronization contract" in the second sentence of design decision 4 should provide the FPGA.

3.6.2 Synchronization Contract between CPU and FPGA applications

The synchronization contract between the CPU application and its counterpart on the FPGA specifies how the two cooperate during their execution to ensure the correct functionality. There does not exist a one-size fits all approach for all C3-aware applications. The contract depends mainly on how the FPGA receives data to process and consequently transfers to the CPU. This informs which notification method is appropriate and if the CPU needs a way to apply backpressure to the FPGA.

This seems very abstract and vague. Hence, let us look at two examples.

Encryption Offloading In this setting, the CPU application offloads the encryption of data to a C3-aware encryption application on the FPGA which transfers the ciphertext back to the CPU using direct cache injection. Each encryption is initiated by the CPU with a pointer and size to a plaintext. The encryption application can also pipeline multiple encryption requests. A possible contract is that whenever the CPU requests a new encryption, it is immediately ready to receive a notification because it wants to achieve the lowest possible latency. Thus, it selects the synchronous UND notification method when registering with the C3.

An alternative scenario might be that the CPU will issue multiple encryptions over the course of its execution and eventually handles all the notifications at once. Because of this it selects the buffered file notification method. In this situation, the contract comprises that the CPU will not request too many large plaintexts to be encrypted such that the ciphertexts still fit in one way of the L2 cache and it does not request more encryptions than it has space in the notification buffer before handling the notifications.

Smart-NIC In this setting, the FPGA is configured as a C3-aware smart NIC which offloads all protocol processing and delivers the received data to a CPU application using direct cache injection. To coordinate reading and writing from a buffer, this synchronization contract sets up a ring buffer where the updates to the written pointer from the FPGA are delivered using C3 notifications and the updates of the read pointer from the CPU are written to a register in the smart-NIC. Because the written pointer of a ring buffer increases the area where the CPU application may read from, the smart-NIC uses Notify data instructions to only send the address and the CPU application uses the file notification method which drops the old notification. With the update of the read pointer—or rather with the lack of an update—the CPU can signal backpressure to the smart-NIC.

3.7 Summary

In this chapter, we looked at the design choices and tradeoffs for the design for the C3 system. The system consists of the Cache Control Coprocessor (C3) isolated on a dedicated CPU core and implemented as a pinned Linux kernel thread and the C3 envoy, an FPGA component maintaining a queue of C3 instructions and providing a register which the C3 continuously polls to fetch new instructions. The C3 executes the fetched instructions which provide functionality for prefetching cache lines into the CPU L2 cache, enabling direct cache injection from the FPGA, cache management instructions, notifications, and SGIs. We investigated several key mechanisms in detail: how can we reliably prefetch cache lines into the ThunderX L2 cache, how can we employ cache allocation for the ThunderX L2 cache and how useful could it be for C3-aware applications, and how we can best isolate the C3 on a dedicated core. Further, we saw

the encoding and description of all C3 instructions. Lastly, we looked at the interface to C3-aware applications with registration and notification instructions and a discussion about the synchronization between the CPU and FPGA part of an application using the C3.

4 Implementation

In this chapter, we look at the implementation details and challenges of the C3 system as it is implemented for this work.

4.1 Infrastructure

Before implementing the C3 system itself, a fair amount of implementation was required to setup some infrastructure to enable deeper introspection into Enzian required for testing (e.g. to investigate prefetching as seen in section 3.3.1) or debugging the C3 system.

4.1.1 Getting as suitable Linux Kernel

Originally, the plan was to implement the C3 as a kernel module written in Rust since the Rust for Linux project is upstream as of Linux 6.2 [13]. However, the golden images for Enzian are based on Ubuntu 20.02 with a 5.4 kernel. Therefore, building a custom kernel was on the order.

Compiling a new Linux kernel is the easiest when it is done on the system the kernel to be compiled will be running on. This allows to use `make(1)` targets like `make localmodconfig` which generates a minimal kernel configuration based on the configuration of the running system and only enabling modules currently loaded, and it simplifies the generation of the initramdisk[1], greatly. However, Enzian is booted using TFTP images of the file system. These images have a size limit of 5 GB, which poses a challenge when you need to install new compilers and other tools. Because the minimal requirements to build a kernel with Rust need a newer version of `clang` than Ubuntu 20.04 offers, we install a newer version of `clang`. But because this is a separate package, we now have two versions of `clang` installed. Together with other needed updates, the system will be larger than 5 GB and we have to start over because we end up in a state, where most temporary files cannot even be created anymore. By first uninstalling big packages, we can give ourselves

[1]The initramdisk is a minimal root file system used during the boot process of Linux. It is mounted as the root file system when the system first boots and its main job is to mount the "real" root file system.

some wiggle room to not run out of space. Luckily, the Rust toolchain can be installed in the scratch directory using `rustup` and the kernel can be compiled there as well. The last remaining challenge is to build the initramdisk, which needs a load of scratch space to copy modules, often causing the system to run out of space resulting in having to start again[2]. This can be remedied by mounting a RAM filesystem since Enzians feature loads of DRAM. This odyssey has been documented on the Enzian Wiki[3] for future students wanting to compile their own Linux kernel on Enzian.

Sadly, after some experimentation with the kernel running Rust, implementing the C3 in Rust proved to be infeasible as the API in the upstream 6.6 kernel was still quite limited and lacked needed features like kthreads. Since maintaining patches to the Linux kernel is not the goal of this work, C3 is now a kernel module written in C. However, we still need a custom kernel, because we need some special configuration options and some APIs which the 5.4 kernel does not export. We are building C3 as an out-of-tree loadable kernel module. But out-of-tree modules can only use symbols explicitly exported in the kernel tree using the `EXPORT_SYMBOL` macros[4]. For instance `kthread_create_on_cpu()` has only been exported since version 5.17 of the Linux kernel. As for configuration, we want some more debug functionality to gain some introspection if needed and some additional functionality. Our config disables all `CONFIG_STRICT_DEVMEM` rules, allowing userspace to directly map physical memory even in if it is not device memory, and it enables `CONFIG_RCU_NOCB_CPU` to enable the offloading of RCU callbacks from certain cores.

In order to be able to quickly boot with our custom kernel with additional commandline parameters, the `emg` had to be modified. Before, we had to edit the bootloader configuration every time we wanted to boot an Enzian with a different kernel than the golden images. By implementing flags[5] to specify the kernel image, the initramdisk, and additional kernel commandline parameters when invoking `emg-acquire(1)`, we can now boot with our custom kernel as follows:

```
emg acquire -n $IMAGE_NAME -k mhaessig/vmlinuz-6.7.1-enzian-debug \
    -i mhaessig/initrd.img-6.7.1-enzian-debug \
    -a "isolcpus=nohz,domain,managed_irq,47 nohz_full=47 rcu_nocbs=47 \
    rcu_nocb_poll" zuestoll09
```

[2]Good thing Enzian has 48 cores, which lowers compilation times significantly.

[3]see https://unlimited.ethz.ch/display/enzianwiki/Compile+a+Custom+Linux+Kernel+for+Enzian

[4]Some APIs are also only exported if the out-of-tree module is GPL licensed which is why the C3 module is BSD/GPL dual licensed.

[5]Commit: https://gitlab.inf.ethz.ch/project-openenzian/enzian-boot-and-image-management/-/commit/a737009dda5e3c1f2c8cbb4ed49d9f53e5106180

4.1.2 Enzian FPGA Driver

The Enzian project provides a Linux driver that maps memory of the FPGA DRAM address space into some processes virtual address space using `mmap(2)`[6]. This memory driver maps FPGA memory using 1 GiB huge pages, by implementing only the huge page handler. Unfortunately, this driver does not compile on a newer 6.7 kernel, as it uses methods which are not exported and which changed in the meantime[7]. Thus, an FPGA memory driver had to be implemented.

Our new driver is a bit simpler as it uses `remap_pfn_range()` to map a range of pages based on their physical address. However, using `remap_pfn_range()` also prevents us from supporting huge pages or prefaulting as Linux allows this only for memory it considers RAM. On Enzian, the FPGA DRAM memory is not given to Linux to be used as RAM, as Linux would use it in its allocator because its support for NUMA and according memory placement restrictions is still quite limited. Additionally, the driver provides resource information for the FPGA DRAM and I/O address spaces to other kernel modules wanting to map FPGA memory. A Linux resource struct is basically a range and a tag that marks this range as an address range of memory. Further, the driver also provides a user header to provide the constants for the location and size of the FPGA's address spaces.

4.1.3 Enzian ThunderX Driver

The preexisting Linux driver for FPGA memory also provides a handful of `ioctl`s to provide access to CvmCache instructions, which are normally only accessible from EL1. Since we need more functionality related to the ThunderX and because it is not really related to FPGA memory, we implemented a separate driver for the ThunderX on Enzian. This driver provides mainly `ioctl`s to expose functionality in EL1 to programs in user space. Further, it provides headers for userspace programs with definitions for `ioctl` numbers, system register addresses and layouts.

System Registers

As in all ARMv8 CPUs, most system registers are only accessible from EL1 or higher using the `msr` instruction. Because we want to give userspace programs deep introspection into the CPU, we implement `ioctl`s for reading and writing from a select set of system registers. Note, that these `ioctl`s are only needed for core system registers, also called AP

[6] Repository: https://gitlab.inf.ethz.ch/project-openenzian/enzian-software/linux-memory-driver

[7] The driver was reimplemented shortly before the deadline for this work and now also works on newer kernels. Also, it now uses only exported symbols.

registers on the ThunderX. All other configuration and status registers on the ThunderX are memory mapped and can be accessed by mapping the appropriate memory regions.

Apart from the core system registers described in the ARM ARM [39], the ThunderX also features a set of Cavium specific registers, easily identified by their `AP_CVM*` names [43]. Among these Cavium specific registers are `AP_CVM_ACCESS_EL{1,2,3}` which can configure access for certain Cavium specific features to lower exception levels. They configure access to CvmCache instructions and different groups Cavium specific core system registers. Because we want to enable introspection from EL0, the goal is to be able to configure EL0's access from the driver we are currently describing.

To do this, we must first ensure that Enzian's ARM Trusted Firmware (ATF), the EL3 monitor, is set correctly, so we do not trap into EL3, because our settings in `AP_CVM_ACCESS_EL1` are overridden by those made by the ATF in `AP_CVM_ACCESS_EL{2,3}`. As it turned out, the ATF only enabled the CvmCache instructions on EL1 and disabled the rest, which is very reasonable for a firmware distributed to commercial customers as these registers offer very low-level access into the system and do pose a security risk if exposed unwittingly. But since Enzian is a research-computer, the firmware received an update which zeroes the relevant bits in `AP_CVM_ACCESS_EL{2,3}` such that EL1 can fully configure all access provided by `AP_CVM_ACCESS_EL1`.

To simplify the process for userspace programs, the driver provides a Cavium library to request access to the different primitives provided by `AP_CVM_ACCESS_EL1` using the read and write system register `ioctl`s. Further, this library also exposes the shared headers with the definitions of the system register bitfield structs and print functions for select registers to help with the parsing of the register contents. To prevent unnecessary system calls into the kernel, the Cavium library tracks the permissions requested by userspace programs and depending on the permissions performs a direct call to read a system register or execute a CvmCache instruction in EL0 instead of issuing an `ioctl` at the cost of some state and a branch. This library can be seen in action in experiment 3.1.

L2 Cache Inspection

While it is possible to determine if a read hit in the L2 cache or not using timing, it would be nice to get a definite answer by reading the state of the cache line in question. Luckily, the ThunderX offers the L2 Cache Index Load Tag CvmCache instruction which loads the tag of an L2 cache line specified by its set and way into one of the `L2C_TAD(0..7)_TAG` data registers [43].

All L2C registers are memory mapped and can be accessed by userspace if the appropriate address space is mapped using `/dev/mem`. We provide a shared header with definitions of CvmCache instructions, constants, addresses, and bitfield struct definitions for registers. The L2C userspace library uses this header to implement functions which, given a physical

address will return the state of the cache line, or simply if the address hits in the L2 cache.

Because the Index Load Tag instruction takes an L2 cache index and the way, we have to read the cache line at the specified index in each way of the L2 cache. If a cache line is not invalid and the tags match with the specified address, we have found a hit. Because this procedure to determine if a cache line is in the L2 cache is not atomic, it can return false negatives, although only very rarely the L2 cache is highly contended.

For CPU-homed cache lines, we can also find out in which state they are held on the FPGA by fetching the remote tags of a cache line. Each DC has to track the cache line states of its homed cache lines on both the home node and the remote node. However, currently this is not implemented in our library.

Most of this functionality was first implemented by Jasmin Schult over the course of her master thesis [54]. This library is an extension of her implementation. The library described here can be seen in action in experiment 3.2.

TLB Inspection

Similar to the L2 cache, we also want introspection into the ThunderX's TLBs. Again, the ThunderX provides a CvmCache instruction to read each level of the TLB hierarchy: µTLB read, MTLB read, and WCU read. All three instructions write the mapping tag and data into the AP registers `AP_CVM_XLATVTAG{0,1}_EL1` and `AP_CVM_XLATDATA{0,1}_EL1`. Thus, to check if a virtual address hits in one of the TLBs, we have to read each entry in the TLB and read one core system register for every entry in the worst case.

Reading the µTLB will thus often evict the mapping we want to check from the µTLB due to this large overhead. This is demonstrated by experiment 4.1.

Following along: **Experiment 4.1: µTLB Observation**

Required bitstream: None
Script: `experiments/utlb-observation.sh`
What you will see: The script repeatedly reads from an address and then reads the µTLB to see if the just accessed address hits.

Unfortunately, the output obtained from reading an entry of page walker cache can not be interpreted yet because the documentation only provides us with abbreviated names of register fields. These field names make sense for the µTLB and MTLB, but not for the walker cache due to its different nature of not caching last level translations.

Software Generated Interrupts

The ThunderX's GIC provides 16 SGIs with interrupt IDs 0 through 15. In its GIC initialization code, Linux allocates the SGIs 0 through 7 to the IRQ numbers 1 through 8[8] as inter-processor interrupts (IPIs), but not the upper eight. The comments state that the SGIs 8 through 15 are often used by firmware[9]. Checking the Enzian ATF, we find that it does not allocate any SGIs.

This leaves the problem of allocating SGIs 8 through 15 with Linux after the GIC has been initialized. We can simply do what the GICv3 driver on Linux does for the lower SGIs at initialization for the higher SGIs at initialization of the ThunderX driver[10]. Linux allocates the SGIs 0 through 7 as per CPU IRQs. We can reimplement the functionality by copying some internal structs into our module and reimplementing the functionality of some non-exported functions. With per CPU IRQs, we need to enable every IRQ on every core to which we will send it. Otherwise, we get "unexpected IRQ" warnings from the kernel. Using `request_percpu_irq(irq, handler_func, &cpu_number)`, where `cpu_number` is a per CPU constant defined in `linux/smp.h`, to register an interrupt handler, an SGI to all cores will be handled on every core separately if we enabled the `irq` on every core. While this behavior is desired for IPIs where one core signals mostly exactly one other core, it is not desired in all cases for C3. While there might be usecases for IPI-like behavior, it is more important to ensure that only one interrupt handler handles an SGI if they are used as notifications.

Thus, we allocate the upper SGIs as "normal" rising edge triggered interrupts[11]. To be able to allocate IRQs, we need a pointer to the IRQ domain and the `struct irq_chip` of the GIC. We can obtain this from the `struct irq_data` of an SGI allocated by Linux. It turns out that the SGIs 0 through 7 have IRQ numbers 1 through 8. Therefore, we can `irg_get_irq_data(1)` and obtain the IRQ domain and chip struct from there. With this, we can create new mappings for SGIs 8 through 15, set an IRQ flow handler and chip information, and clear the `IRQ_NOREQUEST` flag, so we can in fact request the SGIs.

To be able to handle SGIs we need to send them in the first place. Luckily, this is as easy as writing to the GIC register `ICC_SGI1R_EL1`. This generates a non-secure group 1 SGI [40]. An SGI can be sent to all cores, or to a group of 16 cores specified in a CPU

[8] On Linux the IRQ number 0 does not exist.

[9] See https://elixir.bootlin.com/linux/v6.7/source/drivers/irqchip/irq-gic-v3.c#L81

[10] If you, dear reader, ever find yourself in the position of reading the SMP init functions of a GIC driver to reimplement its functionality, please take the following advice. The drivers use a `struct irq_fwspec` to pass parameters to the IRQ allocation function. For example in `gic_smp_init()` in `irq-gic-v3.c` Linux allocates the lower 8 SGIs. Note, that `param_count` is 1, but no parameter is specified. Of course, since we want to allocate SGI 0 through 8, the struct initialization will take care of setting the invisible to 0 implicitly. Remember this and save hours of your life. Don't ask me how I know.

[11] The implementation with the per CPU allocation can still be found on the sgi-percpu branch of the C3 Kernel Module repository.

mask and affinity routing. Our driver implements a slightly more ergonomic wrapper around the Linux provided function to write to the interrupt generating register.

To enable other kernel modules to request the upper eight SGIs, we export a resource struct with the IRQ numbers. In order to test our freshly allocated, we can use the sgi-test kernel module which installs a handler for SGI 8. Then handler logs on which core it was called using `trace_printk()`, because plain old `printk()` cannot be used in an interrupt handler as it might block. See experiment 4.2 for a demonstration.

***Following along:* Experiment 4.2: SGI Test**

Required bitstream: None
Script: `experiments/sgi-test.sh`
What you will see: The ThunderX driver allocates SGIs 8 through 15 in its probe and the SGI test module installs a handler for SGI 8. After issuing an interrupt on SGI 8 the trace shows that the interrupt was handled.

If you run experiment 4.2 multiple times, your Enzian will freeze at some point. Hence, our implementation has a mistake. While time and debugging priorities did not permit an in depth investigation, at this time the IRQ flow handler seems a likely culprit as it might not acknowledge the SGI properly. Further, on a second run of experiment 4.2 we find three handled interrupts in the trace. This might be a hint that SGIs must be treated as per CPU IRQ regardless. In any case, more investigation is needed.

4.1.4 Enzian Memory Explorer

Experimenting with different low-level memory accesses over ECI on Enzian using C programs quickly becomes tedious. Opening files, mmaping memory, fiddling with pointer offsets, translating virtual addresses to physical addresses because some instructions take physical addresses while others take virtual addresses. These papercuts make the development of a test error-prone and leads to frequent recompilation which in turn takes a lot of time. All this ceremony is only prescribed by the method of the testing and not due to the thinking. Mostly, we come up with these tests by thinking of a series of operations on different locations in memory. On Enzian these memory locations often end up being physical addresses, because the documentation talks about physical addresses and the FPGA talks about physical addresses or L2 cache indices and tags, which again map to physical addresses.

To remove the ceremony from experimenting on Enzian the *Enzian Memory Explorer* is a simple read-evaluate-print-loop (REPL) to perform operations on physical addresses. It maps the entire FPGA DRAM address space and parts of the CPU DRAM and FPGA I/O address space. The latter two are mapped using `/dev/mem` which requires a kernel configured with `CONIFG_DEVMEM=y` and `CONFIG_STRICT_DEVMEM=n`. Every instruction with an

address then computes if the address is within a mapped address space and uses the offset into this address space to get a pointer to the desired memory location. The Enzian Memory Explorer implements most functionality mentioned above (see the help message in appendix A).

Because the memory explorer was implemented in an evening after a day of debugging fueled frustration it does not use `readline(3)` as any sane REPL would, but it is entirely handwritten. Despite that it features editing and a history. To avoid a dependency on ncurses it also needed an implementation of `getch()` to receive all keypresses unbuffered, including control characters for arrow keys and the like. It relies on disabling canonical mode and echoing of input characters over the `termios(3)` interface. This enables fully controlling the command line and manipulating the cursor using ANSI control sequences. The history is just a simple circular buffer with 100 entries. It does not duplicate a command in the history if the previous command was the same as the current command and otherwise always appends the last command at the end. Despite not using `readline(3)` the history and editing work just fine and only very esoteric keypresses will be ignored to not scramble the REPL.

Apart from the REPL mode of operation, the memory explorer also supports scripting simply by adding a list of quoted commands on program invocation. This allows for simple reproducible test scripts without any ceremony. This feature is used in this thesis in most follow along experiments.

While the quick testing and scripting enabled by the memory explorer is a huge time and nerve saver, it is not a full replacement for proper C programs as tests. The instructions executed by the CPU always depend on the command input. Due to this fundamental fact we have a data dependency on every access which prevents potential reordering of instructions. While data effect reorderings due to ARM's weak memory model, are technically still possible, the amount of work performed between two commands it becomes very unlikely. Thus, tests exploring reordering on the ThunderX are probably not served well by the memory explorer.

With the REPL processing and command parsing between every two commands the tested sequence of instructions is not really what the user intends. This is mostly fine because only the memory explorer is using the memory behind its mappings, but some operations relying on the architectural state of the CPU like `l2-line-state` or `utlb-hit` are affected by the additional work. For instance, the result of experiment 4.1 is also influenced by this effect, but the observation remains valid, as it also occurs with a C program. Also, because of the overhead the Enzian Memory Explorer should not be used for measurements.

4.2 Cache Control Coprocessor

The Cache Control Coprocessor (C3) is implemented as an out-of-tree loadable Linux kernel module. It consists mainly of setup code which starts the C3 fetch and execute loop on the core isolated using the `isolcpus` command line parameter and the code related to notifications. It relies on the FPGA and ThunderX drivers described above to be loaded for mapping memory and SGIs.

4.2.1 Setup & Core Isolation

During module initialization C3 creates a misc device and uses the device struct to allocate the large C3 context data structure. In C3 we use devres [29] to allocate resources as it takes care of unmapping all resources allocated using devres, which alleviates memory leaks and a bunch of boilerplate code at the cost of a few bytes of overhead. Further, FPGA memory for the registers of the C3 envoy and all of the FPGA DRAM address space are mapped. After initializing the notification subsystem, the C3 starts the C3 envoy by writing a one into the enable register. Afterwards, we check if the enable register actually reads one and that the sanity register contains the expected value. These two additions were very helpful in diagnosing bugs during development. Only after starting the C3 envoy the C3 starts the C3 loop by creating a kthread on the CPU core passed as an argument to the module and then calling the loop function on that kthread. By starting the loop after the envoy, we ensure that it will always read an instruction from the envoy and do not need any checks in the loop to ensure that the envoy is ready.

Note, that starting the C3 kernel module activates the C3 envoy on the FPGA. Conversely, removing the module stops the C3 loop and subsequently the C3 envoy. With this we ensure that one lever enables the entire system with all needed components and relieve the user of the responsibility to correctly orchestrate the startup.

The heavy lifting of isolating the core is performed by the command line parameter `isolcpus`, which is described in detail in the design investigation section 3.3.3. The kernel documentation suggests tweaking the affinity of the IRQ that were not able to be moved off of the isolated core. While this could in principle be done by iterating over all active IRQs and removing a bit from the affinity mask of each IRQ descriptor, we again run into the problem of symbols which are not exported. We can also achieve the same from userspace by writing the new affinity mask to `/proc/irq/$IRQ/smp_affinity`. Experimenting with this showed that all IRQs were already removed from the isolated CPU core or the affinity of the IRQ cannot be modified.

Investigating tasks still running on the isolated core reveals that a handful of kernel threads for workqueues are still affine to it. The C3 could iterate over all tasks known

to Linux to find these kernel threads and move them off of the isolated core. However, currently we do not do this due to lack of time.

4.2.2 C3 Loop

At its core the C3 loop is an endless fetch, decode, execute loop. However, before every iteration it checks if it was signalled to stop by the module exit function. Fetching an instruction is a straightforward read to the instruction register of the C3 envoy, which might stall the read but always send an instruction so the C3 does not "notice". Right after the fetch we first check if the instruction is zero in which case there was no instruction ready. If no instruction is ready, the loop short circuits, increments a counter to track the amount of NOPs received between instructions and continues with the next iteration.

On arrival of a "proper" instruction the C3 decodes the opcode using a macro which clears the most significant 57 bits. If a debug variable is defined at compile time, the instructions get printed in the kernel log with their arguments based on their form. Based on the opcode, the C3 has one big `switch` block where all instructions are fully decoded and executed.

Non-ranged instructions like SGIs or notifications, simply decode the rest of the arguments and call the appropriate function with the supplied arguments. Ranged instructions like fetch and lock need to execute an instruction in a loop. Intuitively, one would write a for loop to execute a fetch and lock for the entire range of cache lines. This, however, creates a data dependency on the very first iteration which will always be done because ranged C3 instructions operate on at least one cache line. But the compiler cannot infer that the length is always at least one because it is loaded from some volatile memory location it cannot assume anything about. By using a do-while loop instead we express this fact to the compiler, which eliminates a branch before the first fetch and allows it and the CPU to pipeline the fetch with the bitfield read to decode the length. Further, cache management instructions like fetch and lock need a `dsb` to become visible to the system. When executing a bunch of these instructions in a loop we can put a single `dsb` at the end of the loop because the cache management instructions do not rely on the visibility of each others effects. We can see all this in the decoding and execution of the fetch and lock C3 instruction in listing 4.1.

In effect, the C3 loop is an infinite loop running in the Linux kernel which never yields. Thus, it hogs CPU time and starves all other tasks on the C3 core, which is exactly why we isolated the C3 in the first place. However, the Linux kernel is designed to run short running tasks or longer running tasks which cooperatively yield so it can schedule all tasks fairly according to their priority. Because the isolation is not complete, running the C3 in testing caused errors in the kernel due to starved tasks. As stated in the previous section, there are some possible improvements for the isolation we did not get to. But as

```
switch (c3_opcode) {
  case C3_OPC_FETCH_LOCK: { // form 0
    u64 base_cl_paddr = c3_cache_line(raw_instr);
    u16 len = c3_len_minus_one(raw_instr);
    u64 cl_idx = 0;
    do {
      u64 cl_paddr = base_cl_paddr + (cl_idx * L2_CL_BYTES);
      cvmcache_l2_fetch_and_lock(cl_paddr);

      cl_idx += 1;
    } while (cl_idx <= len);
    dsb(sy);
    break;
  }
  ...
}
```

Listing 4.1: Example of the decoding and execution of a ranged C3 instruction.

a workaround, the C3 loop yields to the scheduler after every 100 NOPs which prevents all starvation errors encountered previously. While this goes against the goal of a low latency fetch loop, it is better to have a stable system than an unstable but lightning fast system.

4.2.3 Notifications & Interface to User

Notifications are handled off of the isolated C3 core. But before the applications can receive a notification, the C3 has to send them. As per design decision 10 we have a unified data format for notifications which we implement such that it fits in 64 bits (see listing 4.2). The structure is a `union` which includes different representations used in the delivery of notifications. The C3 fills the data according to the notification instruction it received and calls the `send_notification()` function with the notification data and the application ID.

For an application to receive a notification it must register itself using the registration `ioctl` provided by the C3. The C3 has a fixed number of slots available for notifications. In its context struct each notification slot has a cache line to store all state for a notification. C3 maintains a mapping from of indices to applications IDs in a separate array. During registration it checks that the application ID, which can be any 8-bit number to reduce the possibility of conflicts, does not conflict with any ID already in use. If not the new ID is registered at a new index and the notification state at the chosen index is initialized for the requested notification method. Once the application unregisters using the unregister `ioctl` the C3 simply removes the application ID from the mapping.

```
union c3_ntfy_data {
    // This is a duplicate of "data", but identifying with `identify.format`
    // makes for much more understandable code.
    struct {
        c3_ntfy_format format : 8;
        __u64 ignored : 56;
    } __attribute__((packed)) identify;
    struct {
        c3_ntfy_format format : 8;
        __u64 length : 14;
        __u64 cache_line : 42;
    } __attribute__((packed)) cl_range;
    struct {
        c3_ntfy_format format : 8;
        __u64 data : 56;
    } __attribute__((packed)) data;
    __s64 i;
    __u64 u;
    char c[8];
};
```

Listing 4.2: C3 notification data format.

The notification state (see listing 4.3) is shared between the sending and delivering side of the C3 and thus memory shared across cores. Every notification method has a different layout in the tagged union. The file based methods rely on Linux waitqueues to signal ächross cores. The overwrite and drop file based notification methods use the same principle: A delivered flag indicates if the previous notification has been delivered to the application, i.e. the application has read the associated data. If the delivered flag is set to 1, then the sending side can safely write the associated data as the receiving side will not read it because it is waiting for new data, i.e. the delivered flag being set to 0. All reads and writes follow acquire/release ordering. The buffered file based notification method relies on a cache line sized lock-free ring buffer for buffering of notifications. UND, unfortunately, has not been implemented yet due to time constraints.

Because the notification state is shared memory and because any two applications are most likely on different cores, aligning the notification state to one cache line is important to prevent cache line thrashing. Further, all accesses to the cache line state is lockless wit the fewest writes possible on the receiving side such that the state remains in the L1d cache of the C3 if possible.

With this background we can get back to sending a notification from the C3 using `send_notification`. Given the notification data and the application ID, the C3 can obtain the index and thus the notification state. Using the notification method indicated by the notification state it then sends the data provided by the notification instruction.

```
struct c3_notification {
    union {
        struct {
            wait_queue_head_t wq;
            atomic64_t data;
            // Indicates if the previous notification was delivered (1) or not (0).
            atomic_t delivered;
        } file;
        struct {
            wait_queue_head_t wq;
            struct c3_ntfy_rb rb_data;
        } file_buffered;
        struct {
            void *und_cl;
        } und_sync;
    } kind;
    // Tag for the union above.
    enum c3_ntfy_method method;
    u8 id;
} __attribute__((aligned(128)));
```

Listing 4.3: The C3 notification state.

The reception of a file based notification is started by a `read(2)` or `poll(2)` syscall on the C3 device file used to register the application. In the kernel, C3 first checks if there is already a notification ready to deliver (remember, file based notification methods allow for asynchronous notification delivery). If not, C3 sleeps on the waitqueue until a notification is ready and then delivers the notification. `poll(2)` does not deliver notifications carrying data, but only notifys the application that a notification is ready for delivery using the `POLLIN` return value. It does, however, deliver notifications without associated data which is signalled using `POLLIN | POLLPRI` as a return value. This complicates the receiving end slightly, requiring a compare exchange loop, but obviates a second system call.

When using SGIs as notifications, C3 provides a cache line of shared memory for SGI vectors. Whenever the C3 receives an SGI instruction it writes to the vector at the index corresponding to the SGI hardware interrupt ID with release ordering before issuing the SGI. An interrupt handler can then read from the vector, which is exported as a symbol, with acquire ordering.

Additionally, the C3 provides an `ioctl` to adjust the amounts of cycles the C3 envoy stalls a read to the instruction register when no instruction is available. It is mainly for testing purposes (see section 5.1).

4.3 C3 FPGA Envoy

The C3 envoy[12] is a fairly straightforward FPGA component. It consists of an AXI-lite subordinate providing read and write access to a handful of registers and an instruction FIFO. The only complexity is that one register reads from the reading end of the instruction queue and stalls reads if the queue is empty.

The envoy provides a readable and writable enable register. This signal is exposed as a pin to other FPGA components to ensure that they will not try to use C3 while it is not running. Further, it provides a read-only sanity register which returns a known value and a read-write stall cycles register with which an application can set the number of cycles the envoy should stall reads to the instruction register when not instruction is available.

Finally, a read to the read-only instruction register causes one C3 instruction from the instruction queue to be returned to the reader (usually the C3). If the queue is empty, the envoy starts incrementing a wait counter every cycle until it reaches the number of stall cycles configured using the corresponding register. If the instruction FIFO is still empty by that time, the envoy returns a no-instruction C3 instruction, which is all zeros. However, every cycle while stalling the envoy checks if there is now an instruction in the queue. If an instruction appears in the queue it is immediately returned.

The C3 envoy FPGA component is built such that it can be included in the project of an FPGA application as a git submodule. It provides a `create_ips.tcl` script, which should be added into the project creation script used in the workflow of Enzian FPGA projects.

For debugging, the C3 envoy provides introspection using integrated logic analyzers (ILAs). It features generic parameters to configure whether to use the ILAs when the component is instantiated.

Currently, the instruction queue of the envoy is a Xilinx FIFO queue intellectual property (IP). However, when a congested design has to run in a different clock domain (the C3 envoy runs on the same clock as ECI) the design has to add an asynchronous FIFO in front of the already existing queue for clock domain crossing. These duplicate queues are unnecessary and so it would be nice to have an asynchronous FIFO as the instruction queue which can perform the clock domain crossing at the interface to the envoy. This allows the AXIlite interface to the ECI I/O-bridge to run in the same clock domain as ECI which is desirable since we want to be accurate with stalling the reads to avoid timeouts.

[12]Link to repository: https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2023-msc-mhaessig/c3-fpga-envoy

4.4 FPGA Test Applications

To test the C3 system incrementally, several test applications of increasing complexity were developed. While they are not examples of useful practical applications of C3, they serve as tools to explore the capabilities and problems of the system as well as tools for characterizing the system.

4.4.1 No Instruction Test

The C3 no instruction test FPGA component[13] is basically the Enzian static shell with the application stub and the C3 envoy as shown in figure 4.1. This component will never send anything besides a NOP. This component was used in initial testing to ensure the AXIlite subordinate implementation of the C3 envoy is working. Further, it found use in determining the effects of stalling instruction reads on the rest of the system (see section 5.1).

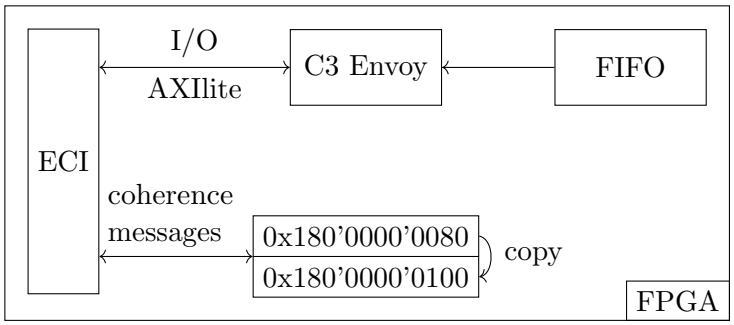

Figure 4.1: Schematic of the no instruction test.

4.4.2 Fixed Instruction Test

The C3 fixed instruction test FPGA component[14] extends the no instruction test with a fixed instruction loop as shown in figure 4.2. Its main use was for initial testing of the C3 with instructions. The instruction loop has an array of C3 instructions and roughly every 10 seconds it sends the next instruction from that array, wrapping around back to the first instruction once it reached the end of the array. Due to its inflexibility its use was only limited to the very first earliest testing of the C3. Further, testing to fetch memory also relies on the cache line copy mechanism. In terms of synchronization, the CPU

[13]Link to repository https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2023-msc-mhaessig/c3-fpga-no-instruction-test
[14]Repository Link

application must always be ready for a notification or select an appropriate buffering notification method as it has no possibility of feedback to the fixed instruction test.

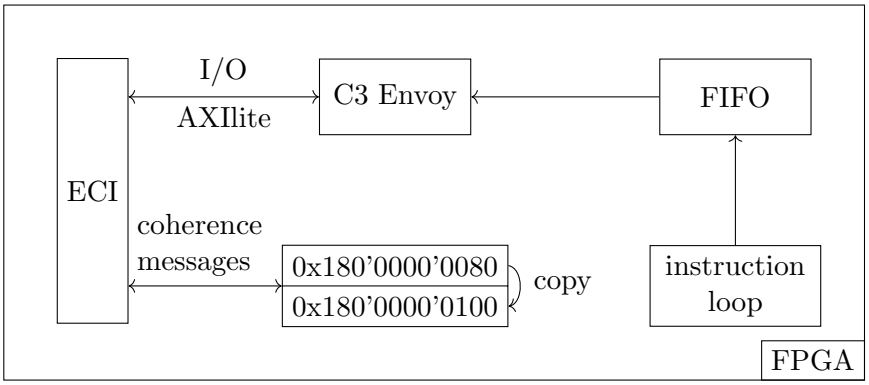

Figure 4.2: Schematic of the fixed instruction test.

4.4.3 Dynamic Instruction Test

Due to the flexibility needed the fixed instruction test involved to the C3 dynamic instruction test FPGA component[15]. The dynamic instruction test provides registers in the FPGA's I/O address space to enable the component and to provide the C3 instructions the component should issue. The application provides instructions by first setting the array instruction length register and then writing the instructions in the instruction array registers. The component provides has a maximum instruction array length of 32 instructions. The CPU application is responsible that it writes correct C3 instructions as they are not checked by the FPGA component. The dynamic instruction test iterates through the instruction array like the fixed instruction test. It also provides a register to set the number of cycles the test should wait between sending instructions. Because we now have two components on the FPGA which need to connect to the ECI I/O-bridge over AXIlite, we need an AXIlite interconnect to demultiplex the requests to the two different segments of the address space as shown in figure 4.3.

As for synchronization between the FPGA and CPU applications, the CPU application, much the same as for with fixed instruction test, has no way of giving feedback to the FPGA application. Thus, it must either be ready to receive an instruction all the time or use a buffering notification method.

The dynamic instruction test is, like the two test applications before, only useful for testing and debugging the C3, however in a more flexible way due to its reconfigurability. It also relies on the cache line copying component for cache line transfers. As such, tests

[15]Link to repository: https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2023-msc-mhaessig/c3-fpga-dynamic-instruction-test

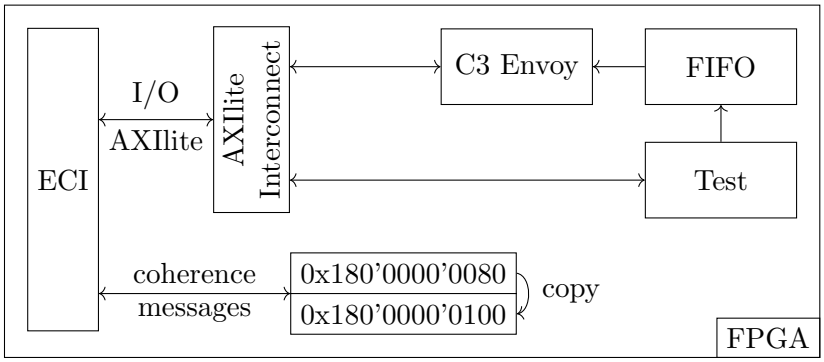

Figure 4.3: Schematic of the dynamic instruction test.

with the dynamic instruction test still do not rely on the FPGA to perform coherent writes independently of CPU writes.

4.4.4 Copy Word Test

While the following test is not yet implemented because the idea only came up during the writeup of this thesis, it is the basis for the simplest kind of useful C3 aware application. The C3 copy word test waits for a write to a register and then proceeds to copy the written value coherently into a FPGA-homed cache line and issues cache injection and notification C3 instructions for this cache line. It is designed to measure the roundtrip latency of a 64-bit word using C3 for the return trip with a bare minimum of processing on the FPGA.

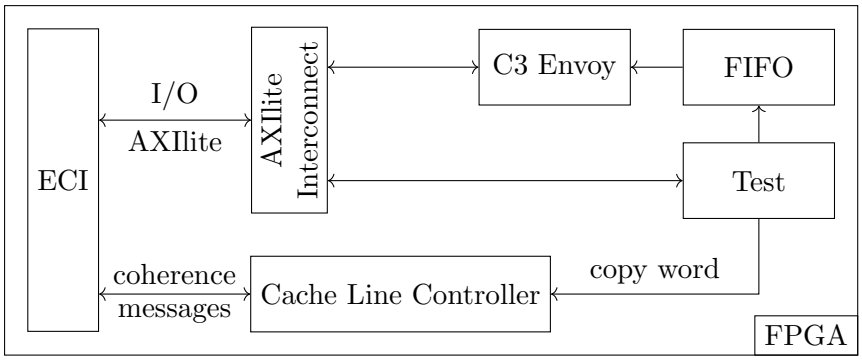

Figure 4.4: Schematic of the copy word test.

As we can see in figure 4.4, the schematic is almost the same as for the dynamic instruction test. The copy word test would provide a register to enable its state machine and a register to write a word which should be copied. In the copy word's state machine in figure 4.5 we see that the enable register works as an active low soft reset and a write to

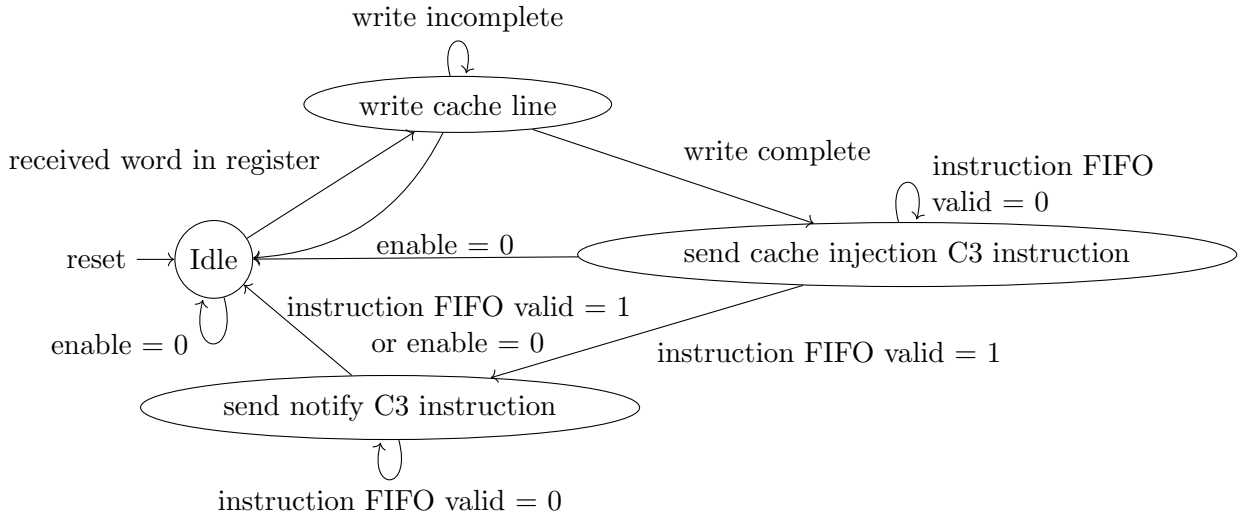

Figure 4.5: State machine of the copy word test.

the copy register triggers the rest of the state machine. To get a comparison between a coherent read from memory and direct cache injection we could add a register which contains a bypass flag. If the flag is set the state machine would skip the sending of the cache injection C3 instruction.

Because the FPGA is only acting upon a CPU request synchronization between the two is given and the CPU must thus only be ready for a notification after sending a request to the FPGA.

The copy word test is useful beyond testing, because it can serve as the basis for actual C3 applications. While the copy word test only performs a copy, an application could perform arbitrary processing after such a request. The single cache line controller is a much simpler alternative to the DCS if only few cache lines are needed. While the application shown here only features one cache line, it can easily be extended to multiple cache lines. As an untested rule of thumb we conjecture that if it is known in advance that an FPGA application will never need to return more than 20 cache lines to the CPU, it should use multiple single cache line controllers instead of the DCS. This should significantly reduce the complexity of the design and reduce the amount of effort needed to reach timing closure on the design.

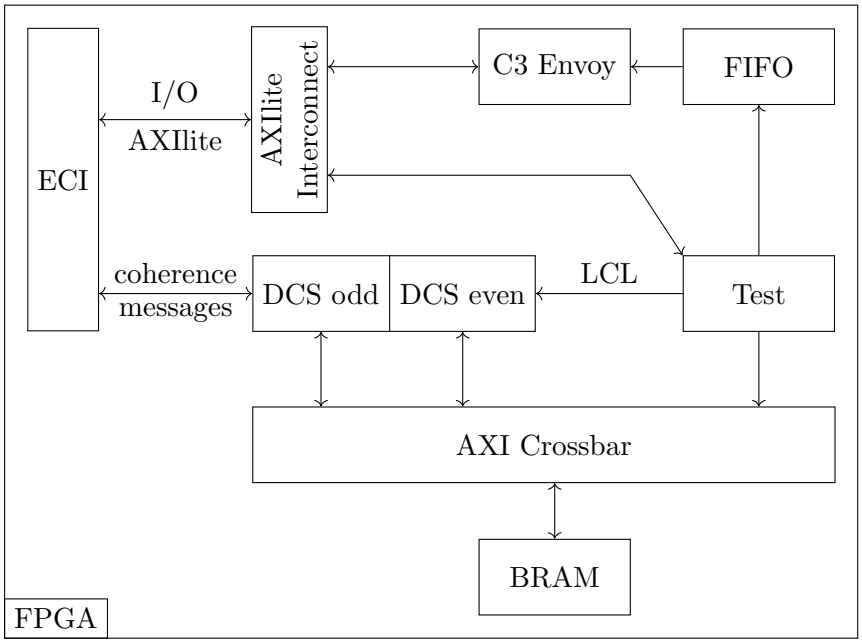

Figure 4.6: Schematic of the DCS copy word test.

4.4.5 DCS Copy Word Test

Before the idea for the copy word test materialized, the DCS copy word test [16] was created. It was created with the same end-to-end latency benchmark in mind, but using the DCS to extend the test for transferring multiple cache lines in the future. The DCS copy word test follows more or less the design overview from figure **??**.

Figure 4.6 shows the schematic of the DCS copy word test. In principle, it replaces the cache line controller from the copy word test with the DCS. The schematic does not accurately show that the local interface to the DCS is actually demultiplexed from the test component to the odd and even DCS'. As a consequence, we have to issue a LCI request to the correct DCS — odd cache lines to the even DCS and vice-versa — and wait for the LCIA to ensure that the cache line we want to write to is written-back and invalidated in the CPU cache so we can write to it. Writing to the BRAM over the AXI crossbar has to be done in two parts, because the DCS is only able to write half a cache line at a time over its AXI interface. After writing to the cache line, we also have to issue an UL request so the CPU can access the cache line again. This increase in complexity leads to the state machine shown in figure 4.7. Note, that if the DCS copy word test is disabled after a cache line was invalidated, it will have to be unlocked before reaching

[16] Link to repository: https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2023-msc-mhaessig/c3-fpga-dcs-return-word

the idle state. Otherwise, the next request might fail because the cache line is already invalidated.

The DCS copy word test can also be extended to transfer more than one cache line by introducing a counter and an edge from send notify to send LCI as long as the counter has not reached the desired value. Such a loop can also be pipelined quite nicely. Further, to get the difference in latency due to direct cache injection, we can simply skip the state where the cache injection C3 instruction is sent based on a register value.

The synchronization between the CPU and the FPGA is ensured by the fact that the DCS copy word test only progresses out of the idle state upon receiving a request from the CPU. Thus, the CPU knows to expect a notification right after issuing a request.

The principle of the DCS copy word test is the foundation of any larger C3 aware application. Instead of simply copying a 64-bit word we can perform any processing and use a very similar state machine to send the data to the CPU.

Challenges

With the increased complexity of the DCS also came a lot of challenges. For instance, when multiplexing the AXI interface of the two DCS' one does not only need to be aware that these AXI interfaces are using cache line indices as addresses, but that these indices are shifted to the right by one. That is because the DCS AXI interface to memory was designed to be used with one memory bank for each odd and even slice. Hence, the last bit of the cache line index would always be the same because all cache lines in the memory bank would be either odd or even. To ensure that half of the memory bank is not wasted, the last bit was removed. However, that bit is needed when multiplexing the interfaces using an AXI crossbar, lest we alias half of the FPGA DRAM address space.

Another challenge, not just with the DCS copy word test are Xilinx IPs. They are tedious to configure, because they must be configured using TCL scripts and not using generic parameters. Further, the AXI interfaces in Xilinx IPs is often not compliant to the AXI specification [1]. For example, it is perfectly legal to keep the ready signal high between two requests if a component is immediately ready again, but many Xilinx components require that after a valid handshake ready is pulled low for at least one cycle. Some components also have stricter ordering requirements between the address and data handshakes than the specification requires. For that reason, the Xilinx BRAM generator was switched out for Alex Forencich's AXI RAM [25] which features a much better AXI implementation and configuration over generic parameters directly in hardware description language (HDL).

The most challenging problem with the DCS copy word test is achieving timing closure. To fill an entire cache way with data, the design goal is to have 1 MiB of low-latency

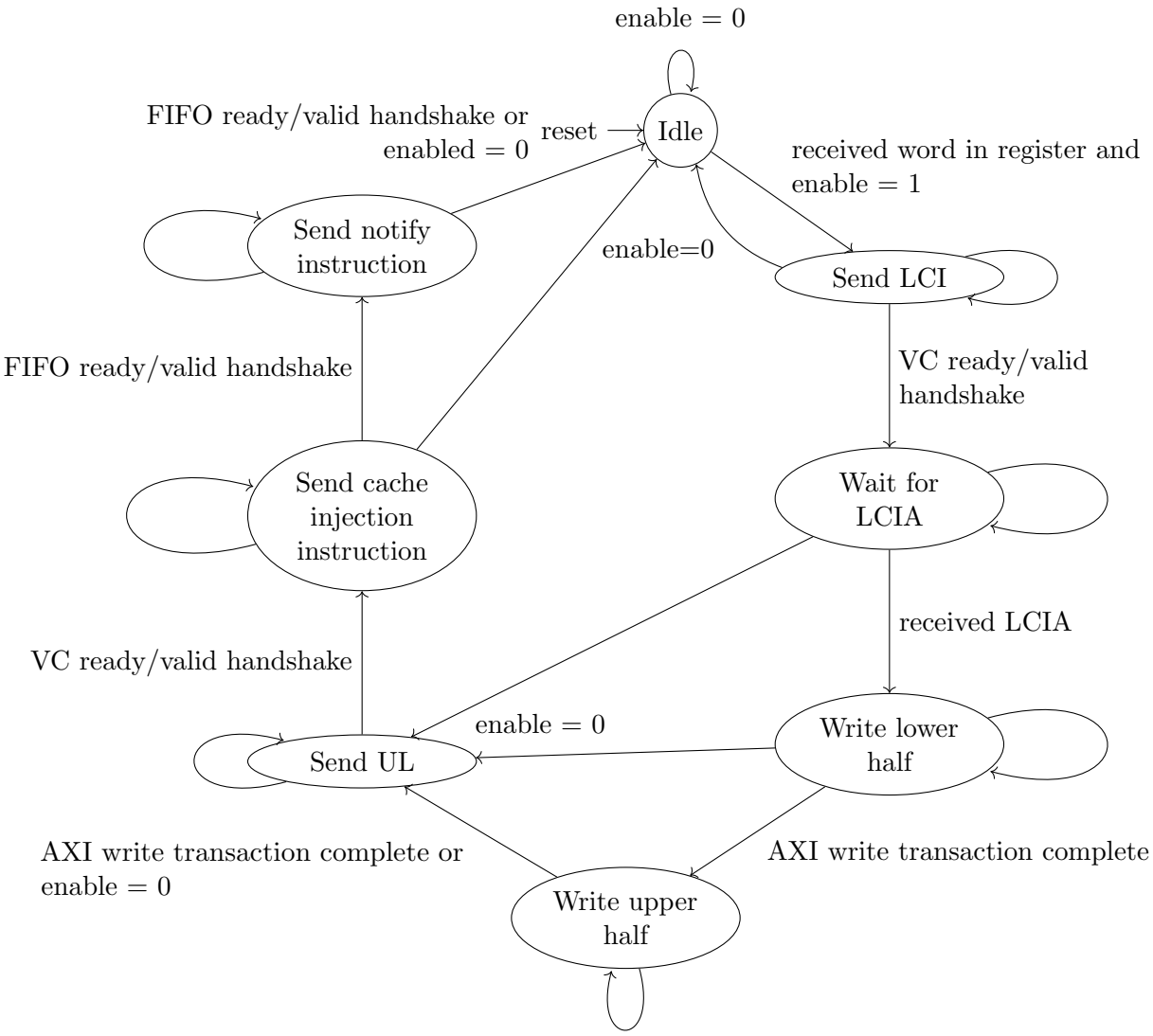

Figure 4.7: State machine for the DCS copy word test.

memory — in this case BRAM. Now, 1 MiB of BRAM is quite a lot so the initial thought was that this contributes to the congestion in the design together with the ILAs. Since the ILAs were needed for debugging initially, the amount of BRAM was reduced to 512 cache lines, which is an eight of an L2 cache way. However, this reduction did reduce resource usage, but not the congestion.

As it turns out, the DCS is a very congested (congestion level 5 out of 6 according to Vivado) design. This makes it almost impossible to meet timing. The best worst negative slack (WNS) ever achieved for the implementation of the DCS copy word test is about -0.25 ns. If a design does not meet timing this is the equivalent of undefined behavior in compiled languages as not all data will be ready before the next clock cycle and the computation will then produce and propagate garbage. Often this manifests as an asynchronous external exception when trying to run the experiment on Enzian.

The furthest the implementation of this application was tested is that an LCI was issued and an LCIA was received. The test got stuck when writing to the lower half of the cache line due to the Xilinx BRAM generator. However, by accessing the locked cache line from the CPU and getting a timeout error confirmed that invalidating and locking the cache line using the DCS was successful.

The Solution

The solution to the DCS not meeting timing would be to run it at a lower clock frequency to give the data more time to propagate between the clock edges. Currently, the DCS runs at the same clock as ECI, which is approximately 322 MHz. With a WNS of -0.25 ns a clock frequency of 250 MHz should provide ample headroom by increasing the clock period by more than 0.8 ns.

To be able to run the DCS at a lower frequency, we need to introduce clock domain crossing into our design. Because there is no benefit to running only the DCS at a lower clock frequency, we also run the application, the BRAM, and the AXI crossbar at the same lowered frequency. To cross clock domains, we add asynchronous FIFOs to each ECI VC and to the C3 instruction queue. In figure 4.8 the red shaded area shows the new, slower clock domain. Every arrow crossing into or out of the clock domain will need an asynchronous FIFO to enable clock domain crossing.

Unfortunately, due to lack of time, implementing clock domain crossing for the DCS copy word test is out of scope for this thesis.

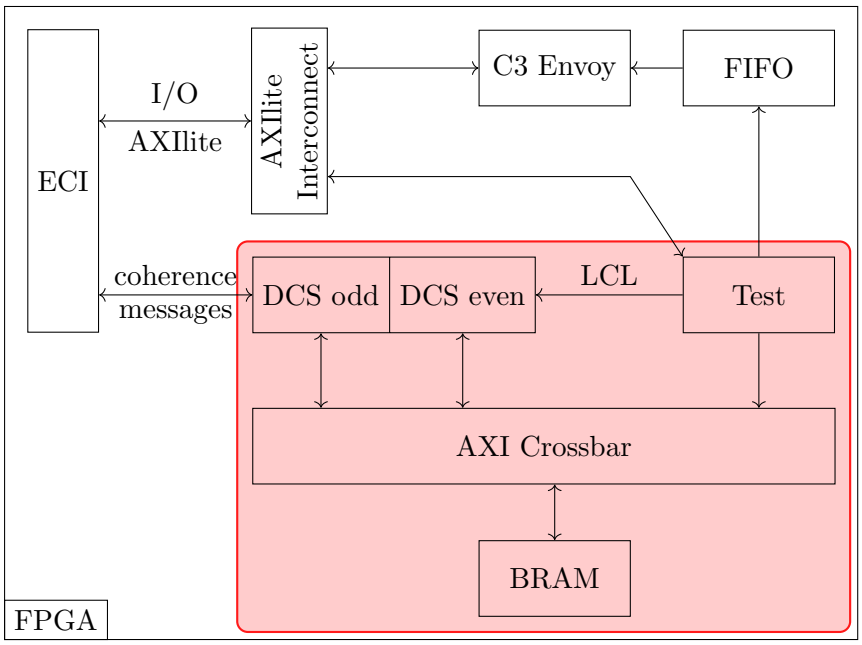

Figure 4.8: Schematic of clock domain crossing for the DCS copy word test.

5 Evaluation

This chapter should evaluate the C3 system with different tests. However, due to the copy word tests being unfinished because of the limited time available for this thesis. Instead, this chapter presents the one test which should be performed to evaluate the system.

5.1 The Influence of Stalling Instruction Fetches on the Rest of the System

The C3 envoy stalls the C3's read when no instruction is ready. Early during testing, the shell would not respond anymore after the C3 was started. This raised the suspicion that the stalling the instruction reads might not only stall the C3 but also influence other cores as a read transaction remains in flight for tens of microseconds. To determine the effect on the rest of the system, we run the C3 no instruction test (see section 4.4.1) on the FPGA, which has the C3 continuously polling for instructions, and run the PARSEC benchmark suite [6] on the rest of the cores.

PARSEC is a benchmark suite for multiprocessors featuring a set of benchmarks "diverse in working set, locality, data sharing, synchronization, and off-chip traffic"[6]. It was originally designed to run on the i386, AMD4, Itanium, Sparc, and PowerPC architecture. Since Enzian features an ARMv8 CPU, we need to use Ciro Santilli's [53] port to Ubuntu 22.04 supporting ARM cross-compilation. Unfortunately, we cannot use the benchmarks `facesim` and `x264` on ARM, because they depend on vector instructions which would require significant porting work. However, porting vector instructions to aarch64 is out of scope for this work. The next challenge is obtaining the inputs to the benchmarks. The PARSEC website has been offline as of November 2023 and only reachable through archive.org[1]. Luckily, archive.org also includes downloads in its snapshots, so we can download the inputs if we have some patience.

The experiment is conducted on `zuestoll07` with all code and input stored on the NVMe SSD. Core 47 is isolated such that the C3 can run on it as described in section 3.3.3. The FPGA is loaded with the no instruction test bitstream (see section 4.4.1). It consists

[1]Working link to the latest snapshot: http://web.archive.org/web/20220930220452/https://parsec.cs.princeton.edu/

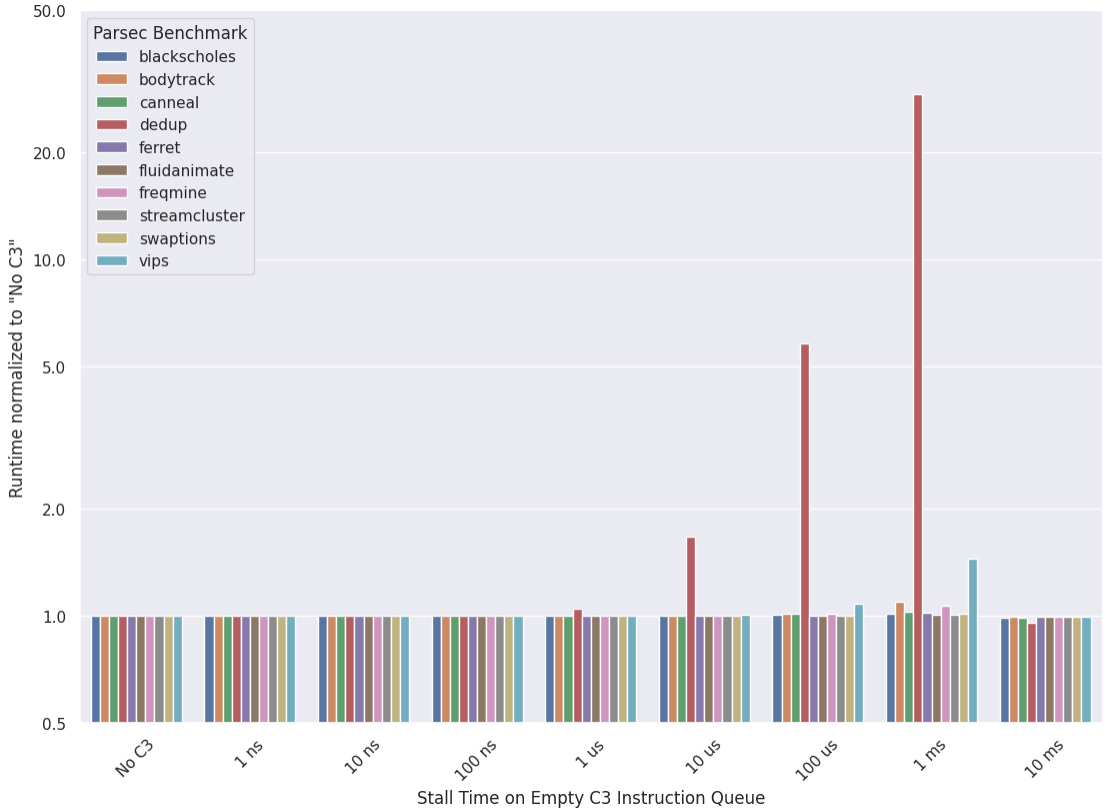

Figure 5.1: Plot of the normalized runtime of PARSEC benchmarks with the C3 running on the system stalling instruction read requests for different amounts of time.

of multiple runs of the PARSEC benchmark suite (without `facesim` and `x264`) with native inputs. The first run consists of only the PARSEC benchmarks without the C3 running. It is thus referred to as "No C3". On the second run, the C3 is running and continually fetching instructions with the C3 envoy configured to stall requests for 1 ns before returning a no instruction available C3 instruction. Before every subsequent run, the C3 envoy is configured to stall for ten times longer, i.e. 10 ns on the second run, 100 ns on the third, and so forth.

Figure 5.1 shows the runtime of the PARSEC benchmarks normalized to each of their runtime without the C3 running for each run with increasing stalling time on the C3 envoy. Hence, the runtimes for each PARSEC benchmark are relative to the corresponding benchmark shown in the leftmost group labeled "No C3". The x-axis shows the different stall times of the C3 envoy during the runs of the PARSEC benchmark suite. Each bar color corresponds to a PARSEC benchmark as indicated by the legend. The y-axis shows the normalized runtime on a logarithmic scale. If a parsec benchmark has a normalized runtime of 2.0, this means that it ran two times slower than the corresponding benchmark without the C3 running.

At a first glance, we see that the higher the stall time, the slower the runtimes of the PARSEC benchmarks tend to be. However, the stall time of 10 ms is an anomaly as it shows times even faster than without a C3 running. This is because the C3 crashed due to an asynchronous external exception in the beginning of the 10 ms run. It did so across three different instances of running this experience, each with slightly different implementations of the C3 envoy. Hence, we exclude the 10 ms stall time from this discussion.

Next, we can see that the benchmarks exhibit noticeably slower runtimes starting with a stall time of 1 µs. We could draw the conclusion from this that the C3 envoy should stall as little as possible or even not at all, because the rest of the system is clearly less affected than if we stall instruction fetches. However, this conclusion would be premature as it does not take the effect on C3 applications into account, because this experiment does not feature any. Whenever the C3 envoy stalls an instruction fetch and an instruction enters the queue the instruction is immediately returned leading to the lowest possible time between the C3 instruction being issued and the instruction being executed on the C3. Since one of the goals of C3 system is low latency, some stalling of instruction fetches should be beneficial. However, this experiments allows us to conclude, that stalling fetches for longer than 1 µs will affect the rest of the system in some cases.

Further, figure 5.1 shows that not all PARSEC benchmarks are affected equally by the C3 envoy's stalling of instruction fetches. Most affected is the `dedup` benchmark with very significant slowdown of 30x for a stall time of 1 ms, followed by the `vips` benchmark with a slowdown of roughly 1.3x for a stall time of 1 ms. While the `bodytrack`, `canneal` and `freqmine` benchmarks also experience some slowdown for a stall time of 1 ms, the rest of the benchmarks are not noticeably affected. Looking at the benchmarks in detail, the only thing which really differentiates `dedup` from the rest of the benchmarks is that it only has a traffic from the cache, i.e. the amount of data transferred to and from the cache, of around 1.5 bytes per instruction which is lower than all other benchmarks with have a traffic of around 3 bytes per instruction or more [6]. Also, the second and third-lowest traffic have the `bodytrack` and `vips` benchmarks at slightly less than 3 bytes per instruction. However, the `canneal` and `freqmine` benchmarks have exhibit cache traffic of 4 bytes per instruction, more than other benchmarks which are not noticeably affected by the stalling of instruction fetches. This suggests that cache traffic is part of the cause for the slowdown but not the whole story.

Looking at off-chip traffic, `dedup`, `bodytrack`, `freqmine`, and `vips` all tend to have more store off-chip traffic with increasing core size. The paper showed the figures for off-chip traffic for 16 cores. Since the ThunderX features 48 cores this effect is likely more exaggerated. The `canneal` benchmark has significant, but slightly decreasing writeback and load off-chip bandwidth needs as the number of cores increase. The `streamcluster` benchmark needs around 0.9 bytes per instruction of off-chip load bandwidth, but is entirely unaffected by the stalling of instruction fetches. This suggests that off-chip traffic

for writebacks and stores are slowed down by stalling non-caching reads issued from the CPU to the FPGA.

This experiment seems to disprove our hypothesis used in the design of the C3 control path that reducing traffic on the memory bus would reduce the effect of continuously polling for C3 instructions. Instead, stalling the non-caching reads seem to cause some sort of blocking on the memory bus of the ThunderX which mainly affects stores. As a consequence, design decision 3 should be revisited with this new knowledge. However, further tests on the benefits of stalling C3 instruction reads on the C3 envoy for the latency of C3 aware applications should be conducted.

5.2 Future Tests

This section discusses tests not conducted due to the necessary FPGA test applications not being completed in time. The following tests should help to characterize the C3 system.

C3 Roundtrip Latency Test Using the C3 copy word test application, this test should determine the roundtrip latency of a 64-bit word sent to the FPGA and then immediately sent back using the C3 and direct cache injection. The value being sent to the FPGA can be the system counter value right before sending, so when the value is returned using C3 it can immediately be compared to the system counter value right after reading the received cache line. This roundtrip latency test should be conducted for each notification method and for both the copy word test and the DCS copy word test to compare between the different system configurations. Further, both test copy word test should implement a register to allow the CPU application to forego the direct cache injection so a comparison to reading directly from the FPGA can be established.

Direct Cache Injection Throughput Test By extending the C3 copy word test applications to write to multiple cache lines, this test should measure the throughput of data transferred from the FPGA to a CPU application using direct cache injection. The CPU test application should request a number of cache lines from the FPGA application, which then transfers the requested number of cache lines using C3, and measure the time from the writing the request to the FPGA until reading the last cache line on the CPU. Dividing the measured latency by the number of cache lines received gives the achieved bandwidth for that request. By repeating this test for both copy word test applications for larger and larger number of cache lines to transfer we can find out for which transfer size direct cache injection becomes worse than just reading from memory directly.

Direct Cache Injection Reliability Test The direct cache injection reliability test should repeatedly perform direct cache injection for a cache line and track the miss rate in the L2 cache after the cache injection using the different cache injection C3 instructions. This test should also be conducted once where only the test is using the L2 cache and once when the L2 cache is contended because of some other memory intensive load running on the system. By combining this with a latency measurement, we gain an additional data point. This reliability test should show which C3 instruction is best suited for direct cache injection in different situations.

Cache Footprint Analysis of the C3 To run with the lowest possible latency, the C3 would ideally always hit in its L1d and L1i cache. This test should measure the number of misses in the L1d and L1i cache of the C3 using the counters of the ThunderX performance measurement unit.

Influence of Stalling C3 Instruction Reads on the Roundtrip Latency This test is the complement to the test in the previous section (see section 5.1) to determine the benefits of stalling instruction fetches in the C3 envoy. This test works like the C3 roundtrip latency test, but it varies the stall time in the C3 envoy in different runs. Ideally, this is done once with only the test running and once with another memory intensive workload running in parallel. These data points in conjunction with those from section 5.1 should then provide an ideal range for the stall time of C3 instruction reads.

6 Conclusion

In this chapter the results of this project and future work are discussed.

6.1 Discussion

This section discusses the shortcomings and tradeoffs of the C3 system and whether it actually reached its goals.

6.1.1 Is C3 a convincing Cache Coherence Protocol Extension?

To function as an extension of the cache coherence protocol interfacing with C3 should actually feel like interfacing with a cache coherence protocol. While the interface for submitting C3 instructions is very similar to sending ECI messages over VCs. However, from the view of an FPGA application the C3 system lacks one critical feature because of design decision 4. An FPGA application does not receive an acknowledgement when a C3 instruction has been executed. This is contrary to ECI and the DCS local interface where transactions taking a long time will be acknowledged or a transaction is more or less instant and is assumed to be completed instantly when the message is submitted (e.g. the UL message of the DCS local interface). This is the only reason why notification instructions are needed and why the C3 even needs an interface to CPU applications which, notably, cache coherence protocols usually lack. With synchronization traditional notification methods in the form of ring buffers or a polled device register would suffice.

While notifications might be a novel feature for cache coherence protocols worthy of investigation in their own right, saying that no acknowledgements for C3 instructions make for a simpler system as stated in section 3.2.1 is probably not true because it increases the interfaces needed for the C3. Design decision 4 did not consider that and it also did not take into account the goal of behaving like a cache coherence protocol.

The notifications have their advantages. For one, they make C3 instructions “fire-and-forget” from the perspective of an FPGA application. This allows the FPGA application to use its processing resources to “more important things” and putting to work the C3 on a dedicated core, which really ought to do some work to justify getting an entire core

to itself. Further, this "fire-and-forget" nature of C3 instructions and the resulting lower coupling to the coherence protocol and hopefully allowing for more parallelization and less synchronization overhead. But this would have to be proven using tests.

Speaking of decoupling from the coherence protocol, another unexplored tradeoff in the C3 system is the low level nature of the C3 instructions and the simplicity of the C3 envoy. All the C3 envoy really does is aggregating instructions in a queue and delivering these to the C3 on the CPU. Currently, a cache injection C3 instruction functions like a notification to the C3 that data is ready to fetch into the L2 cache. The FPGA application has to take care of coherently writing the data to memory it wants to transfer itself. The C3 envoy could also be involved in the execution of C3 instruction and take care of coherently writing data to memory instead of the application. While this would relieve the application of some burden it couples the design of the C3 envoy to coherence components avaliable on the FPGA and would need to have a different implementation depending on whether cache line controllers or the DCS are used. Such a shift of responsibility would also trade off flexibility of the application to write to whatever memory it sees fit for ease of use. As such the current design meets our goals better than the alternative, but for a non-experimental C3 system certain patterns might pay off to be abstracted in such a way.

Another difference to some coherence protocols, including ECI, is that the execution of C3 instructions is inherently serialized. ECI, however, can reorder packets on the bus and features multiple parallel VCs. While this serialization cannot easily be removed from the C3, it results in the C3 always being slower than a cache coherence protocol implementing the features C3 instructions provide as such a protocol will be able to execute some requests in parallel.

6.1.2 Viability as a Platform for Experimentation

A major goal of this thesis is that C3 should serve as platform for experimentation and thus be flexible, modular and extendable such that future research needs are enabled rather than hindered by the system. This goal is mostly achieved as the implementation provides a wealth of infrastructure (see section 4.1) which can be used for extending the functionality of C3 and is useful for use on Enzian otherwise. The code is modular such that a rewrite of all C3 instructions was completed in all of half an hour thanks to only having to change the definitions in a central location. However, the C3 instructions also show a weak point of the modularity: the code for the C3 and the C3 envoy duplicate most of each other's constants. It would be much nicer if these constants could be derived from some central location such that they only have to be changed in one central location instead of both repositories.

The C3 system is inherently flexible as it is agnostic of how it is used by applications. It only provides instructions and poses only few requirements on how they are used.

While this can be a burden on the applications, for instance in terms synchronization (see section 3.6), it also provides freedom to the application which is the tradeoff this thesis is shooting for

6.2 Future Work

This section presents interesting avenues for future investigation around the C3 system.

Finish what was started This work leaves many components of the C3 system unimplemented which makes us unable to run the tests described in section 5.2 needed to characterize the system. The obvious future work item is thus to finish what this thesis started. In order to characterize the performance of C3 in a real setting, it should be incorporated into a proper application like a smart NIC. A smart NIC is a good choice for a first application as there is a bunch of related work to using direct cache access to accelerate NICs. Further, a smart NIC is an I/O device which is fundamentally different from all tests proposed thus far in that the FPGA application will send cache lines and notifications without the CPU application requesting it directly.

Synchronization This implementation of C3 is non-synchronizing. The CPU side just executes whatever the FPGA side has put in the queue and then goes on to fetch the next instruction. The execution of C3 instructions is completely unrelated from the execution of the FPGA application (see section 3.6). We could extend the C3 such that the CPU executes in lock-step with the FPGA application by sending an acknowledgement to the FPGA whenever it is done executing a C3 instruction. As discussed above this would obviate notifications to CPU applications from the C3 because the FPGA application is able to take care of the notification itself. It would be interesting to investigate the tradeoffs between the two approaches and to see if they yield different performance.

How far can we push precognition? One of the advantages of C3 is that it offers a way to take advantage of the prior knowledge an FPGA application has when it transfers data to a CPU application. For one, the C3 enables push prefetching of the data into the CPU L2 cache from the FPGA because the FPGA knows that the CPU application will want to access the transferred data and thus eliminating the (thought to be) compulsory miss. However, in the same way the FPGA application could use C3 instructions to prefetch other data or instructions needed to process the data which it will transfer. But for that it needs to know what to prefetch, as does the CPU application. A CPU application can be built such that relevant pieces of code and data can be referenced using pointers such that we are able to determine their location in memory or we could use static analysis to find locations which would benefit from prefetching. Once the location to prefetch

are known, the FPGA application can either notify the CPU application that it should prefetch some locations which has the advantage that this memory could be prefetched into the L1 cache, or the FPGA application can issue C3 instructions to prefetch these locations if it is told about these locations, e.g. by writing to some registers. If the data the FPGA application has to transfer contains control-plane data, e.g. the header in a network packet, this information can be transferred before the rest of the data so the CPU application can already configure the data path for the incoming data.

Can cache allocation make a difference? In section 3.3.2 we investigate what could be done with L2 cache way partitioning to reduce possible conflict misses in the L2 cache. The conclusion is that it is probably difficult to derive a benefit due to the difficulty in placing cache lines in the appropriate way in the L2 cache. But a cache control coprocessor should at least try. This test should on one hand test if conflict misses are reduced by way partitioning in a contended cache and on the other hand make sure that it does not hurt performance if the application is running by itself on the CPU with way partitioning still enabled.

Does sharing the C3 make sense? The design and implementation of the C3 system always erred on the side of enabling multiple applications to use the C3 at the same time without ever testing it. Future work should run multiple C3 aware applications in parallel and observe how they interact in the C3. Is the ThunderX L2 cache large enough to handle multiple applications using it for direct cache injection? To share the C3 among multiple applications with different runtimes, it would be interesting to have dynamic reconfiguration of applications on the FPGA to allow new applications to be loaded while running other applications.

6.3 Summary

This thesis introduces the Cache Control Coprocessor (C3), a dedicated core on the Enzian CPU fetching instructions from the FPGA to execute on the CPU. It is designed to serve as an extension to ECI allowing us to implement novel features currently not available in cache coherence protocols with the goal of enabling experimentation with features for new cache coherence protocols in hardware.

Based on the goals of creating a flexible and extensible system for experimentation, we designed the C3 system to be capable of injecting FPGA-homed cache lines into the CPU's L2 cache from the FPGA, of issuing interrupts to CPU cores, and sending notifications to CPU applications that data is ready to be read. FPGA applications can submit C3 instructions into an instruction queue managed by the C3 envoy which is an FPGA component providing a register for the C3 to fetch instructions from. If upon an

instruction fetch from the C3 no instruction is available in the queue, the C3 envoy stalls the read for a configurable amount of time to wait if an instruction arrives in the queue. Tests in this work have determined that stalling for longer than 1 µs will slow down other write intensive programs on the CPU.

The C3 itself is a Linux kernel module isolated on a dedicated core using Linux's `isolcpus` commandline parameter. It consists mainly of a tight instruction fetch loop. Upon receiving a C3 instruction from the C3 envoy, the C3 decodes it and executes the corresponding instruction. During the design, we investigated in detail how the ThunderX executes prefetches into the L2 cache to provide a guide for using the cache injection C3 instructions. We also implemented a ThunderX driver which allocates the SGIs with interrupt number 8 through 15 with Linux such that they can be used by C3 instructions.

As an interface to CPU applications, the C3 provides `ioctl`s to register the applications with an ID such that incoming notifications instructions can be routed to the appropriate notifications and using the appropriate notification method. Because the C3 is a Linux kernel module it implements the `read` and `poll` system calls to deliver notifications to applications. C3 notification instruction serve as the main synchronization primitive between a CPU application and its offloading counterpart on the FPGA. It is often used to signal that data has been transferred into the CPU L2 cache which is now ready to be consumed.

We implemented a handful of C3 aware FPGA applications for testing and characterizing the C3 system. However, due to problems during the implementation the applications could not be finished in time to perform the tests to characterize C3.

Bibliography

[1] ARM. *AMBA AXI and ACE Protocol Specification. AXI3, AXI4, and AXI4-Lite*. IHI 0022E. Version E. 2023. URL: https://developer.arm.com/documentation/ihi0022/e (visited on 08/25/2023).

[2] Nilanjana Basu, Claudio Montanari, and Jakob Eriksson. "Frequent background polling on a shared thread, using light-weight compiler interrupts." In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '21. Virtual, Canada: Association for Computing Machinery, 2021, pp. 1249–1263. ISBN: 9781450383912. DOI: 10.1145/3453483.3454107.

[3] Andrew Baumann et al. "The multikernel: a new OS architecture for scalable multicore systems." In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, Oct. 11, 2009, pp. 29–44. ISBN: 9781605587523. DOI: 10.1145/1629575.1629579.

[4] László A. Bélády. "A study of replacement algorithms for a virtual-storage computer." In: *IBM Systems Journal* 5.2 (1966), pp. 78–101. ISSN: 0018-8670. DOI: 10.1147/sj.52.0078.

[5] Brian N. Bershad et al. "User-level interprocess communication for shared memory multiprocessors." In: *ACM Transactions on Computer Systems* 9.2 (May 1, 1991), pp. 175–198. ISSN: 0734-2071. DOI: 10.1145/103720.114701.

[6] Christian Bienia et al. "The PARSEC benchmark suite: characterization and architectural implications." In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. Toronto, Ontario, Canada: Association for Computing Machinery, 2008, pp. 72–81. ISBN: 9781605582825. DOI: 10.1145/1454115.1454128.

[7] Thomas Burd et al. ""Zeppelin": An SoC for Multichip Architectures." In: *IEEE Journal of Solid-State Circuits* 54.1 (Jan. 2019), pp. 133–143. ISSN: 0018-9200. DOI: 10.1109/JSSC.2018.2873584.

[8] Surendra Byna, Yong Chen, and Xian-He Sun. "Taxonomy of Data Prefetching for Multicore Processors." In: *Journal of Computer Science and Technology* 24.3 (May 26, 2009), pp. 405–417. ISSN: 1860-4749. DOI: 10.1007/s11390-009-9233-4.

[9] CCIX Consortium Inc. *An Introduction to CCIX*. White Paper. CCIX Consortium Inc., 2019. URL: https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf (visited on 03/25/2024).

[10] David Cock et al. “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 ’22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 434–451. ISBN: 9781450392051. DOI: 10.1145/3503222.3507742.

[11] CXL Consortium. *Compute Express Link (CXL)*. URL: https://computeexpresslink.org/ (visited on 03/25/2024).

[12] Linux Kernel Contributors. *The kernel’s command-line parameters*. Version 6.7. 2024. URL: https://docs.kernel.org/6.7/admin-guide/kernel-parameters.html (visited on 04/30/2024).

[13] Jonathan Corbet. “Rust in the 6.2 kernel.” In: *LWN* (Nov. 17, 2022). URL: https://lwn.net/Articles/914458/ (visited on 05/05/2024).

[14] Bill Dally. “Power, Programmability, and Granularity: The Challenges of ExaScale Computing.” In: *2011 IEEE International Parallel & Distributed Processing Symposium*. IPDPS ’11. Anchorage, AK, USA: IEEE Computer Society, May 2011, pp. 878–878. ISBN: 978-1-61284-372-8. DOI: 10.1109/IPDPS.2011.420.

[15] William J. Dally, Yatish Turakhia, and Song Han. “Domain-specific hardware accelerators.” In: *Communications of the ACM* 63.7 (June 2020), pp. 48–57. ISSN: 0001-0782. DOI: 10.1145/3361682.

[16] William J. Dally et al. “Efficient Embedded Computing.” In: *Computer* 41.7 (July 2008), pp. 27–32. ISSN: 0018-9162. DOI: 10.1109/MC.2008.224.

[17] Michael Dalton et al. “Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization.” In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. NDSI ’18. Renton, WA, USA: USENIX Association, Apr. 2018, pp. 373–387. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/dalton.

[18] Robert H. Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions.” In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. ISSN: 1558-173X. DOI: 10.1109/JSSC.1974.1050511.

[19] Ulrich Drepper. What Every Programmer Should Know About Memory. In: *LWN* (Nov. 24, 2007). Ed. by Jonathan Corbet. URL: https://lwn.net/Articles/259710/ (visited on 03/15/2024).

[20] Bengisu Elis et al. “Non-Blocking GPU-CPU Notifications to Enable More GPU-CPU Parallelism.” In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. HPCAsia ’24. Nagoya, Japan: Association for Computing Machinery, 2024, pp. 1–11. ISBN: 9798400708893. DOI: 10.1145/3635035.3635036.

[21] Hadi Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling.” In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA ’11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 365–376. ISBN: 9781450304726. DOI: 10.1145/2000064.2000108.

[22] Alireza Farshin et al. “Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-Hundred-Gigabit Networks.” In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC’20. Berkley, CA, USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4.

[23] Daniel Firestone et al. “Azure Accelerated Networking: SmartNICs in the Public Cloud.” In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. NSDI ’18. Renton, WA, USA: USENIX Association, Apr. 2018, pp. 51–66. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[24] Denis Foley and John Danskin. “Ultra-Performance Pascal GPU and NVLink Interconnect.” In: *IEEE Micro* 37.2 (May 10, 2017), pp. 7–17. ISSN: 1937-4143. DOI: 10.1109/MM.2017.37.

[25] Alex Forencich. *Verilog AXI Components*. Feb. 26, 2019. URL: http://alexforencich.com/wiki/en/verilog/axi/start (visited on 05/12/2024).

[26] John Giacomoni, Tipp Moseley, and Manish Vachharajani. “FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue.” In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’08. Salt Lake City, UT, USA: Association for Computing Machinery, 2008, pp. 43–52. ISBN: 9781595937957. DOI: 10.1145/1345206.1345215.

[27] Chris Gregg and Kim Hazelwood. “Where is the data? Why you cannot debate CPU vs. GPU performance without the answer.” In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS ’11. Austin, TX, USA: IEEE Computer Society, 2011, pp. 134–144. ISBN: 9781612843674. DOI: 10.1109/ISPASS.2011.5762730.

[28] John L. Hennessy and David A. Patterson. “A new golden age for computer architecture.” In: *Communications of the ACM* 62.2 (Jan. 2019), pp. 48–60. ISSN: 0001-0782. DOI: 10.1145/3282307.

[29] Tejun Heo. *Devres - Managed Device Resource*. Version 6.8. 2007. URL: https://docs.kernel.org/driver-api/driver-model/devres.html (visited on 05/09/2024).

[30] Seokbin Hong, Won-Ok Kwon, and Myeong-Hoon Oh. “Hardware Implementation and Analysis of Gen-Z Protocol for Memory-Centric Architecture.” In: *IEEE Access* 8 (July 9, 2020), pp. 127244–127253. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3008227.

[31] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. “Direct Cache Access for High Bandwidth Network I/O.” In: *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. ISCA ’05. Madison, Wisconsin, USA: IEEE Computer Society, May 1, 2005, pp. 50–59. ISBN: 076952270X. DOI: 10.1109/ISCA.2005.23.

[32] Intel Corporation. *Intel Data Direct I/O Technology*. URL: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html (visited on 04/11/2024).

[33] Farshad Khunjush and Nikitas J. Dimopoulos. “Hiding Message Delivery and Reducing Memory Access Latency by Providing Direct-to-Cache Transfer during Receive Operations in a Message Passing Environment.” In: *Proceedings of the 2005 Workshop on MEmory Performance: DEaling with Applications, Systems and Architecture*. MEDEA ’05. Saint Louis, Missouri, USA: IEEE Computer Society, Sept. 2005, pp. 41–48. DOI: 10.1145/1152779.1147358.

[34] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. “Do OS abstractions make sense on FPGAs?” In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. OSDI ’20. Berkley, CA, USA: USENIX Association, 2020, pp. 991–1010. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/roscoe.

[35] David A. Koufaty et al. “Data Forwarding in Scalable Shared-Memory Multiprocessors.” In: *IEEE Transactions on Parallel and Distributed Systems* 7.12 (Dec. 1996), pp. 1250–1264. ISSN: 1045-9219. DOI: 10.1109/71.553274.

[36] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. “When Prefetching Works, When It Doesn’t, and Why.” In: *ACM Transactions on Architecture and Code Optimization* 9.1 (Mar. 2012). ISSN: 1544-3566. DOI: 10.1145/2133382.2133384.

[37] D. Lenoski et al. “The Stanford DASH multiprocessor.” In: *Computer* 25.3 (1992), pp. 63–79. DOI: 10.1109/2.121510.

[38] Edgar A. Leon, Kurt B. Ferreira, and Arthur B. Maccabe. “Reducing the Impact of the Memory Wall for I/O Using Cache Injection.” In: *15th Annual IEEE Symposium on High-Performance Interconnects (HOTI 2007)*. HOTI ’07. Stanford, CA, USA: IEEE Computer Society, Aug. 22, 2007. ISBN: 978-0-7695-2979-0. DOI: 10.1109/HOTI.2007.8.

[39] Arm Limited. *Arm Architecture Reference Manual for A-Profile architecture*. Version J.a. 2023. URL: https://developer.arm.com/documentation/ddi0487/ja (visited on 08/23/2023).

[40] Arm Limited. *Arm Generic Interrupt Controller Architecture Specification. GIC architecture version 3 and 4*. Version H.b. Apr. 2024. URL: https://developer.arm.com/documentation/ihi0069/hb (visited on 05/08/2024).

[41] Jiuxing Liu and Bulent Abali. “Virtualization polling engine (VPE): using dedicated CPU cores to accelerate I/O virtualization.” In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS ’09. Yorktown Heights, NY, USA: Association for Computing Machinery, 2009, pp. 225–234. ISBN: 9781605584980. DOI: 10.1145/1542275.1542309.

[42] Clemens Lutz et al. “Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects.” In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’20. Portland, OR, USA: Association for Computing Machinery, 2020, p. 16331649. ISBN: 9781450367356. DOI: 10.1145/3318464.3389705.

[43] Cavium (now Marvell). *Cavium ThunderX CN88XX, Pass 2. Hardware Reference Manual*. Document Number CN88XX-HM-2.7P. 2017.

[44] V. Milutinovic, A. Milenkovic, and G. Sheaffer. "The cache injection/cofetch architecture: initial performance evaluation." In: *Proceedings Fifth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. MASCOT '97. Haifa, Israel: IEEE Computer Society, Jan. 12, 1997, pp. 63–64. ISBN: 0818677589. DOI: 10.1109/MASCOT.1997.567582.

[45] Sparsh Mittal. "A Survey of Recent Prefetching Techniques for Processor Caches." In: *ACM Computing Surveys* 49.2 (Aug. 2016). ISSN: 0360-0300. DOI: 10.1145/2907071.

[46] Gordon E. Moore. "Cramming More Components onto Integrated Circuits." In: *Electronics Magazine* 38.8 (Apr. 19, 1965), pp. 114–117. ISSN: 0748-3252.

[47] Vijay Nagarajan et al. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Comp. by Natalie Enright Jerger and Margaret Martonosi. Found. by Mark D. Hill. 2nd ed. Synthesis Lectures on Computer Architecture 49. Cham, CH: Springer, 2022. ISBN: 978-3-031-01764-3. DOI: 10.1007/978-3-031-01764-3.

[48] Amy Ousterhout et al. "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads." In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA, USA: USENIX Association, Feb. 2019, pp. 361–378. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[49] Daniel Petrisko et al. "BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs." In: *IEEE Micro* 40.4 (July 1–Aug. 1, 2020), pp. 93–102. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996145.

[50] DPDK Project. *Data Plane Development Kit (DPDK)*. URL: https://www.dpdk.org (visited on 03/25/2024).

[51] Abishek Ramdas. "CCKit: FPGA acceleration in symmetric coherent heterogeneous platforms." PhD thesis. Zürich: ETH Zürich, Nov. 20, 2023, p. 296. DOI: 10.3929/ethz-b-000642567.

[52] Abishek Ramdas et al. "ECI: a Customizable Cache Coherency Stack for Hybrid FPGA-CPU Architectures." In: (2022). DOI: 10.48550/arXiv.2208.07124. arXiv: 2208.07124 [cs.AR].

[53] Ciro Santilli. *PARSEC Benchmark*. 2023. URL: https://github.com/cirosantilli/parsec-benchmark (visited on 05/13/2024).

[54] Jasmin Schult. "Characterization and validation of an in-silicon cache coherence protocol implementation." MA thesis. Zürich: ETH Zürich, Apr. 3, 2023. 242 pp. DOI: 10.3929/ethz-b-000630895.

[55] SiFive, Inc. *SiFive TileLink Specification*. Specification. Version 1.9.3. SiFIve, Inc., Feb. 9, 2023. URL: https://www.sifive.com/document-file/tilelink-spec-1.9.3 (visited on 03/25/2024).

[56] Jeff A. Stuart, Michael Cox, and John D. Owens. “GPU-to-CPU callbacks.” In: *Proceedings of the 2010 Conference on Parallel Processing*. Euro-Par 2010. Ischia, Italy: Springer-Verlag, Aug. 31, 2010, pp. 365–372. ISBN: 9783642218774. DOI: 10.1007/978-3-642-21878-1_45.

[57] J. Stuecheli et al. “IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI.” In: *IBM Journal of Research and Development* 62.4/5 (June 2018), 8:1–8:8. ISSN: 0018-8670. DOI: 10.1147/JRD.2018.2856978.

[58] Wen Su et al. “Using Direct Cache Access Combined with Integrated NIC Architecture to Accelerate Network Processing.” In: *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. HPCC ’12. Liverpool, UK: IEEE Computer Society, 2012, pp. 509–515. ISBN: 9780769547497. DOI: 10.1109/HPCC.2012.75.

[59] Dan Tang et al. “DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance.” In: *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. HPCA ’10. Bangalore, India: IEEE, 2010, pp. 1–12. ISBN: 978-1-4244-5659-8. DOI: 10.1109/HPCA.2010.5416638.

[60] Neil C. Thompson and Svenja Spanuth. “The decline of computers as a general purpose technology.” In: *Communications of the ACM* 64.3 (Feb. 2021), pp. 64–72. ISSN: 0001-0782. DOI: 10.1145/3430936.

[61] Jan Nino Walter. “Linux as a universal boot loader for new operating systems.” MA thesis. Zürich: ETH Zürich, Nov. 28, 2022. 55 pp. DOI: 10.3929/ethz-b-000583404.

[62] Minhu Wang, Mingwei Xu, and Jianping Wu. “Understanding I/O Direct Cache Access Performance for End Host Networking.” In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems*. Vol. 6. 1. New York, NY, USA: Association for Computing Machinery, Feb. 28, 2022. DOI: 10.1145/3508042.

[63] Mark Wyse. *The BedRock Cache Coherence Protocol and System*. Specification. Version 1.1. Paul G. Allen School of Computer Science & Engineering, University of Washington, Apr. 5, 2022. URL: https://github.com/black-parrot/black-parrot/blob/master/docs/bedrock_protocol_specification.pdf (visited on 04/10/2024).

[64] Sun Xian-He, Surendra Byna, and Yong Chen. “Server-Based Data Push Architecture for Multi-Processor Environments.” In: *Journal of Computer Science and Technology* 22.5 (Sept. 25, 2007), pp. 641–652. ISSN: 1860-4749. DOI: 10.1007/s11390-007-9090-y.

[65] Xilinx. *UltraScale Architecture and Product Data Sheet: Overview (DS890)*. Version v4.4.1. 2023. URL: https://docs.xilinx.com/v/u/en-US/ds890-ultrascale-overview (visited on 08/24/2023).

[66] Chia-Lin Yang et al. “Tolerating Memory Latency through Push Prefetching for Pointer-Intensive Applications.” In: *ACM Transactions on Architecture and Code Optimization* 1.4 (Dec. 2004), pp. 445–475. ISSN: 1544-3566. DOI: 10.1145/1044823.1044827.

[67] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. “The Yin and Yang of processing data warehousing queries on GPU devices.” In: *Proceedings of the VLDB Endowment* 6.10 (Aug. 2013), pp. 817–828. ISSN: 2150-8097. DOI: 10.14778/2536206.2536210.

[68] Ardhi Wiratama Baskara Yudha et al. “A Simple Cache Coherence Scheme for Integrated CPU-GPU Systems.” In: *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*. DAC ’20. Virtual Event, USA: IEEE Press, 2020. ISBN: 9781450367257. DOI: 10.1109/DAC18072.2020.9218664.

A Enzian Memory Explorer Help Page

```
enzian-memory: REPL to explore all things memory on Enzian

Usage: enzian-memory [options] "[command1]" "[command2]" ...
Without any commands, the program will start a REPL.
Commands passed over the command line must be surrounded with quotes.

The program maps CPU memory from 0x100000000 to 0x1300000000 and FPGA
memory from 0x10000000000 to 0x20000000000 with an identity mapping.
Therefore, all addresses are physical addresses.

Number formats: <addr> and <hex-value> are expected to be in hexadecimal
                <seconds> and <sgi number> are expected to be in decimal

If not all memory is mapped, not all instructions are available.

Options:
  -c  Do not map CPU memory
  -e  Echo commands
  -f  Do not map FPGA memory
  -h  Print this help message
  -i  Do not map FPGA I/O memory

Commands:
  read <addr>                      Read from physical address
  write <addr> <hex-value>         Write to physical address
  dmb                              Data memory barrier
  dsb                              Data system barrier
  l2-inv <addr>                    Invalidate L2 cache line
  l2-inv-wb <addr>                 Invalidate and write back L2 cache line
  l2-wb <addr>                     Write back L2 cache line
  l2-fetch-lock <addr>             Fetch and lock cache line in the L2 cache
  l2-pref <prefetch type> <addr>   Prefetch address into L2 cache
  prefu <addr>                     Prefetch address into uTLB
  tlb-inv-all                  Invalidate all entries in the TLBs on this core.
  l2-hit <addr>                    Check if address is in L2 cache
```

```
 utlb-hit <addr>                Check if address is in uTLB
 mtlb-hit <addr>                Check if address is in mTLB
sysreg <sysreg name>        Read system register; the name is case insensitive
 l2-line-state <addr>           Print L2 cache line state
 print-tlb <tlb name>           Print TLB; the name is case insensitive
 sgi <sgi number>               Send SGI
 sleep <seconds>                Sleep
 echo <text>                    Echo some text
 help                           Print this help message
 exit                           Exit the program
```