Reverse Engineering the Intel Cascade Lake Mesh Interconnect

by

Miles Dai

S.B., Computer Science and Electrical Engineering, Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author .....
Department of Electrical Engineering and Computer Science
May 14, 2021

Certified by .....
Mengjia Yan
Assistant Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Reverse Engineering the Intel Cascade Lake Mesh Interconnect

by

Miles Dai

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The rising core counts of multicore processors have driven the move to more scalable on-chip networks to allow for efficient communication between the cores, memory subsystem, and other processor peripherals. Intel's latest line of Scalable processors introduced the mesh interconnect in the form of a two-dimensional network of rings that reduces the bandwidth and latency bottlenecks of prior ring interconnects. Much is still unknown about the mesh interconnect, and details from the official documentation are sparse. In this thesis, we perform the first in-depth reverse-engineering of the mesh interconnect on an Intel Cascade Lake server. We use performance counters to determine the layouts of the cores on the die and reverse-engineer the traffic scheduling policy for packet routing on the interconnect. In addition, we develop tools to generate and monitor cross-core cache coherence traffic. We then apply these tools to determine the precise conditions required for traffic contention on the network. This information is combined with publicly available documentation and prior work to provide an unprecedented understanding of the new Intel mesh interconnect. Further, this work paves the way for future investigation into the use of the network-on-chip as a potential hardware side channel.

Thesis Supervisor: Mengjia Yan
Title: Assistant Professor

Acknowledgments

My deepest and sincerest gratitude to everyone who has made this thesis possible throughout this strange and tumultuous year.

First and foremost, to my advisor Mengjia Yan: thank you for your unending support and direction through the course of this project, for introducing me to the fascinating world of hardware security, and for helping me realize my potential as a researcher. Your insight, knowledge, and dedication have been an inspiration.

To Riccardo Paccagnella: thank you for the enthusiasm and curiosity you brought to the project, for always being so ready to dive into the code with me, and for being such an encouraging and fantastic role model for me.

Lastly, to my family: Mom, Dad, and Chelsea, thank you for supporting me from afar, for always believing in me, and for instilling in me the confidence and eagerness to tackle hard problems and strive for the best.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The modern computer is a highly parallelized machine. As chip designers face increasing pressure to reduce device power consumption while observing diminishing returns from uniprocessor architectures, multi-core chips have become the solution for increasing processor performance [15]. A server-class processor today can contain dozens of cores on a single die, and even client-class processors are seeing high core counts [10]. With more cores, however, comes the challenge of low-latency inter-core communication. This was originally achieved through the use of a shared bus that all components would listen on. With an increasing number of modules, the bus quickly became a bottleneck as multiple devices contended on the central communication channel. Modern processors instead use a network of connections with switches and routers that are able to divert various data packets towards their respective destinations. This reduces the need for direct connections between modules at the cost of requiring physical connections to be shared and increasing communication latency. Due to the shared hardware, this interconnect construct opens the possibility of various processes influencing one another through the traffic they send.

The necessity of understanding the hardware structures comes from the increased blurring of the line between hardware and software. In an ideal world, a properly-designed instruction set architecture should present a hardware-independent interface for software to interact with. However, in recent years, this interface has proven to be more permeable than expected. In the past decade, a class of security vulnera-

bilities known as side-channel attacks has grown in popularity. Side-channel attacks rely on the implementation of computer systems to steal information rather than on vulnerabilities in the software itself. Typically, execution of a process leaves traces on the hardware that can leak information to other processes using the same hardware resources. Side-channel attacks have already been demonstrated using many types of shared resources such as DRAM [18], caches [17, 19], and even the power supply [37]. It stands to reason that the shared on-chip network can also be used as a side channel. In fact, recent work has demonstrated the feasibility of using an on-chip ring interconnect to exfiltrate data from protected processes [24] on an older-generation Intel processor. It is clear that shared hardware resources present an attack surface that needs to be considered. The newest models of Intel server-class processors introduced a new mesh interconnect which has yet to be well-understood. In this work, we endeavor to examine the new mesh network that Intel has created and reason about its behavior in the hopes that it will lead to a better understanding of the security risks posed by the new design.

One of the greatest challenges of reverse-engineering on-chip networks is the lack of documentation and tooling available to examine these structures. From the manufacturer's perspective, the on-chip interconnect is not a feature that the consumer needs to be aware of. In fact, an ideal network would be entirely transparent to the software. As a result, there is little information available about the hardware implementation and very limited interfaces to interact with the network from the operating system. Successfully reverse-engineering the mesh interconnect requires developing custom tooling and designing precise experiments to infer details about how the new mesh interconnect operates.

To this end, this thesis makes the following contributions. First, we use performance counters to reverse engineer the layout of the mesh interconnect. This includes mapping out the location of each tile on the processor die and reverse engineering the mapping from Linux CPU IDs to physical core locations. We are also able to leverage performance counters to understand the scheduling policy governing the assignment of packets to specific lanes on the interconnect, a level of detail that has not previously

been available for the mesh network. In the process, we provide additional information about sparsely documented features in Intel's performance monitoring system that can be useful for others conducting similar reverse-engineering work. Next, we design tools to efficiently and reliably generate and monitor network traffic, allowing us to analyze the timing characteristics of the mesh. Finally, we apply these tools to precisely determine the conditions required to observe contention on the network, an important step towards developing a practical side-channel attack.

These steps are critical to understanding the potential for using the mesh interconnect as a side channel. In this work, we have presented the most in-depth understanding of the Intel mesh network from publicly available information to date. Further, the results we observed provide strong evidence to show that even on the latest architectures, interconnect side channel attacks are feasible.

Chapter 2

Background

Modern processor design is extraordinarily complex. While many fundamental concepts have remained unchanged for decades, chip designers go to great lengths to extract every potential source of performance optimization. In this section, we provide a summary of modern CPU cache design and the on-chip interconnect. This description is necessarily high-level as it largely details structures that are well-established motifs in computer processor design. More specific details are often unknown from the information released by the manufacturer and need to be reverse-engineered, as we describe in Chapter 3. We end the section with a brief overview of microarchitectural side-channel attacks.

2.1 CPU Cache Architecture

The problem of implementing fast, random-access memory (RAM) has been present since the development of early computing machines. The rise of processor clock speeds has only exacerbated this difference. As a result, processors use small amounts of very low-latency memory to cache the most frequently accessed data in close proximity to the computing core. Today, caches are an essential piece of hardware that have grown in complexity and continue to undergo significant optimizations with each new processor generation. In this section, we describe the over-arching principles that govern cache design.

2.1.1 Memory Hierarchy

Modern high-performance processors employ multiple levels of caches that gradually trade off access latency for capacity. Smaller, faster caches are referred to as "higher-level" caches whereas larger, slower caches are said to be "lower-level". On multi-core machines, the organization of caches among multiple cores is complex. Generally, processors provide two levels of private caches (L1 and L2) for each core and a shared, lower-level L3 cache, otherwise known as a *last-level cache* (LLC). The L1 and L2 caches, being private to each core, are relatively fast, requiring an access time of around 4-12 cycles. The LLC is much larger but requires an access time of 40-60 cycles. All cache accesses, however, are significantly faster than main memory accesses which can take on the order of 200-300 cycles.

Cache Structure Each level of cache in most Intel CPUs is made of a *data array* and a *tag array*. The data array holds the data or code fetched from main memory that is being cached, and the tag array holds the higher order bits of the address that corresponds to the particular data block in the data array. The data stored in the cache is organized into units called *cache lines*, each one of size B bytes. Modern caches are typically *set-associative* which refers to how the cache lines are organized. A set-associative cache consists of S sets and W ways. Each cache set can hold W cache lines. Figure 2-1 shows an example of a set-associative cache with $W = 4$ and $S = 16$.

Each address maps to a particular cache set based on its address bits. In general, the lower $\log_2 B$ bits indicate an offset into the cache line to a specific byte – the *block offset*. The next $\log_2 S$ bits make up the *set index* and indicate which set the address should be stored in. The remainder of the address bits form the *tag*, which is stored in the tag array to track the address currently stored in the cache line.

Handling a memory request When a particular cache receives a memory request for address A, it will extract the set index bits and check all W cache lines at that index in parallel to see if any of the tags currently in the cache match the tag of A. If

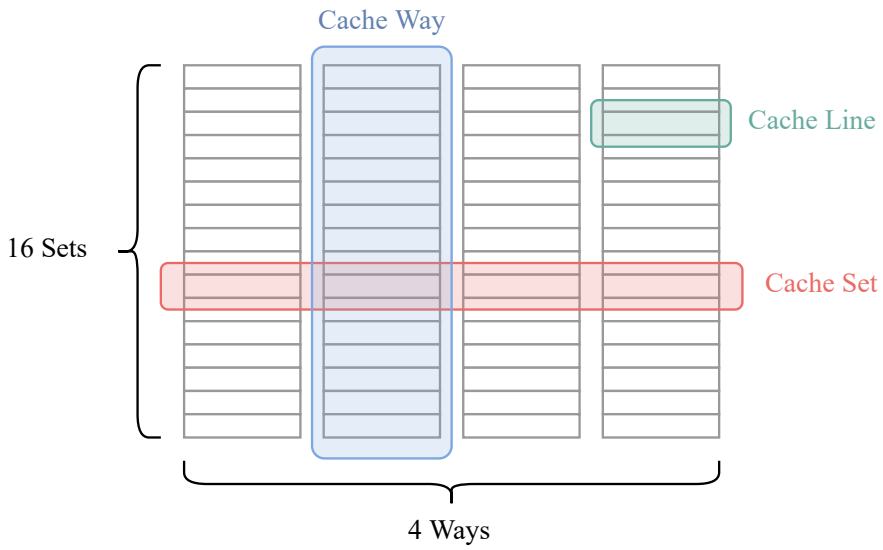

Figure 2-1: Example set-associative cache with 16 sets and 4 ways.

a match is found, then A is present in the cache. We say that the request for A was a *cache hit* and return the data immediately. Otherwise, the data must be fetched from a lower-level cache or DRAM in what is known as a *cache miss*. Further, this data should be inserted into our cache. The cache will look at the corresponding set and insert the address into an empty slot, or it must use a *replacement policy* to decide which address should be evicted to make room for A.

Replacement Policies The choice of which cache line to evict in the event of a cache miss comes down to the replacement policy and, in conjunction with the nature of the workload, can have a significant impact on the cache hit-rate. A common replacement policy is known as *Least-Recently-Used (LRU)* which states that the address which was used farthest in the past should be evicted. LRU takes advantage of temporal locality, the observation that addresses accessed recently are more likely to be accessed again. In practice, perfect LRU is difficult to implement and may be replaced by approximations such as pseudo-LRU or even more intelligent replacement algorithms. Regardless of the details, a replacement policy should, in general, aim to keep more commonly-used lines in the cache.

2.1.2 Multi-core Cache Organization

The LLC of a modern Intel processor is typically partitioned into multiple slices [9]. On client-class processors, there is a one-to-one correspondence between the number of cores and the number of slices. On server-class processors, it is possible for there to exist more LLC slices than cores (as discussed in Chapter 3). This organization allows for a more modular implementation of each core instance and creates a more scalable core design overall.

This sliced design introduces another aspect to how addresses are mapped to specific cache lines. In particular, an address now also needs to be mapped not just to a specific LLC set but also to an LLC slice. This second mapping is often done with a hash function that attempts to distribute addresses evenly across the different slices. In Intel processors, the details of this hash function are undisclosed [6]. Prior work done on older Intel processors reveals that the slice hash function can be expressed as a simple XOR of selected bits [35]. However, more recent work on Skylake microarchitectures suggests that the hash function may be more complex [22].

Cache Inclusivity The LLC can be *inclusive*, *exclusive*, or *non-inclusive* of the private L1 and L2 caches. An inclusive LLC includes all entries in the private L2 caches. An exclusive LLC will not contain any entries present in L2 caches. An entry will not be present in both an L2 cache and the LLC at the same time. For a non-inclusive cache, an entry that is present in a private L2 cache may or may not be present in the LLC. Non-inclusive cache hierarchies have recently become more common. For example, the most recent Skylake and Cascade Lake server processors by Intel use noninclusive caches [9].

Directories and Cache Coherence When multiple cores require access to the same cache line, the caches must be kept coherent to prevent the use of stale data. This is typically accomplished by assigning a state to indicate whether the line is shared, modified, invalid, etc. Most modern processors use a directory-based protocol that uses point-to-point communication to maintain this state. In a directory-based

protocol, a directory structure is used to keep track of which cores contain a copy of a given line in their caches, and whether the line is dirty or clean in those caches. Yan et al. perform a detailed reverse-engineering of the non-inclusive directory implementation and demonstrate that it is more complex than that of previously-used inclusive directories and can be used to construct side channels [32].

2.2 On-Chip Interconnect

The advent of multi-core computing introduced the problem of cross-core communication. While many computational tasks can be parallelized, cores still need to communicate with one another to synchronize processes and maintain coherent caches. Hardware is required to connect cores, caches, on-chip memory controllers, and other peripherals.

On processors with very few cores, a single bus shared by all cores is possible. With more cores, this technique quickly became infeasible as the performance is bottlenecked by contention for access to the bus; some form of an on-chip network is required [15]. Analogous to macroscopic computer networks, on-chip networks consist of data packets being sent through a system of interconnects and routers. Routing algorithms determine the paths taken by these packets. However, unlike computer networks, these routing algorithms are working with much lower latencies, often on the order of single clock cycles, and are frequently implemented in hardware, making them relatively simpler. Also, because the number and configuration of nodes on a processor is static and generally small, routing algorithms can make assumptions about the topology of the network.

2.2.1 Mesh Topology

The choice of topology plays a critical role in the scalability and performance of a microarchitecture. For example, Intel uses a mesh topology for their scalable processors (e.g. Xeon-Phi, Skylake-SP, Cascade Lake), forgoing their more common ring topology due to scalability concerns [11]. With all cores connected on a single ring,

the addition of another core potentially increases the memory access latency and reduces the available bandwidth for all other cores. Intel initially mitigated the issue by dividing the die in half and creating a ring within either half. The scalability issue was addressed more robustly in the Skylake-X/SP microarchitectures with the introduction of the mesh interconnect.

The topology of an interconnect governs the connections between the various hardware modules on a processor die. We refer to these hardware modules as *tiles*. In its mesh topology, Intel arranges these tiles into an array and connects each array to its immediate orthogonal neighbors. Tiles on the inside of the array have four connections to their north, south, east, and west neighbors. Tiles on the edges of the array may have fewer than four connections.

2.2.2 Tile Construction

The mesh interconnect is designed to be modular, allowing various types of tiles to interact with interconnect. Tiles use a shared interface with the mesh which provides greater flexibility in the placement of components and a routing protocol that is agnostic to the type of tile a packet is traversing.

Tile Types Multiple types of tiles can be found on the mesh, incorporating computing cores, memory controllers, and other peripherals. The majority of the tiles are computing cores, which will refer to as *core tiles*. Core tiles each contain one Intel core, private L1 and L2 caches, and an LLC slice. The LLC slice includes the data, directory (snoop filter), and a control unit called the *Caching/Home Agent (CHA)*. According to Intel's documentation [6], the CHA's role is two-fold. As a caching agent, it is responsible for receiving requests to the local LLC slice from other cores or sockets. As a homing agent, it is responsible for issuing requests to remote cores for memory accesses that missed in the private caches and the local LLC slice. Further, the CHA is responsible for maintaining cache coherence between the various cores. In multisocket systems, the CHA will also interact with Intel's Ultra-Path Interconnect (UPI) to send packets.

The other tile type of significance to our reverse-engineering efforts is the *Integrated Memory Controller (IMC) tiles*. The function of the memory controller is to interface with DRAM by translating read and write instructions into specific memory commands [6]. Cascade Lake processors have two IMC tiles on each die which are positioned symmetrically on the east and west edges of the die as shown in Figure 2-2. The positioning of the IMCs on either side of the die minimizes the average latency to each of the cores.

Other tiles such as UPI, PCIe, and DMI interfaces are present along the north edge of the die as shown in Figure 2-2. These tiles can inject and receive data packets from the mesh, allowing for high-bandwidth access to the cores.

2.2.3 Interconnect Design

Rings

Contrary to its name, the mesh interconnect is actually composed of a collection of ring interconnects. As shown in Figure 2-2, all tiles along a row are connected via a ring interconnect and similarly for the columns. Intel's documentation [6] states that the messaging protocol occurs over four rings: address (AD), acknowledge (AK), block (BL), and invalidate (IV). Each ring performs a specific function[1] in the cache coherence protocol.

Address (AD) Carry the memory address of a cache operation during core read-/write requests.

Acknowledge (AK) Provide an acknowledgement back to the sender of a packet that an operation has completed.

Block (BL) Carry data. Each cache line (64 bytes) requires 2 transfers.

Invalidate (IV) The CHA can invalidate particular cache lines when the data for a

[1]This is not an exhaustive list of each ring's function. The Uncore Performance Monitoring guide also suggests that the rings are used in cross-socket communication through the Intel UPI [6]. These have been omitted as they are not relevant to the scope of our reverse-engineering efforts.

cache line is no longer up-to-date such as when another core has modified that particular cache line.

Packets traveling along the ring follow a Y-X routing scheme in which the packet first traverses the vertical ring until it reaches the correct row, at which point it switches onto the horizontal ring to reach its destination [21].

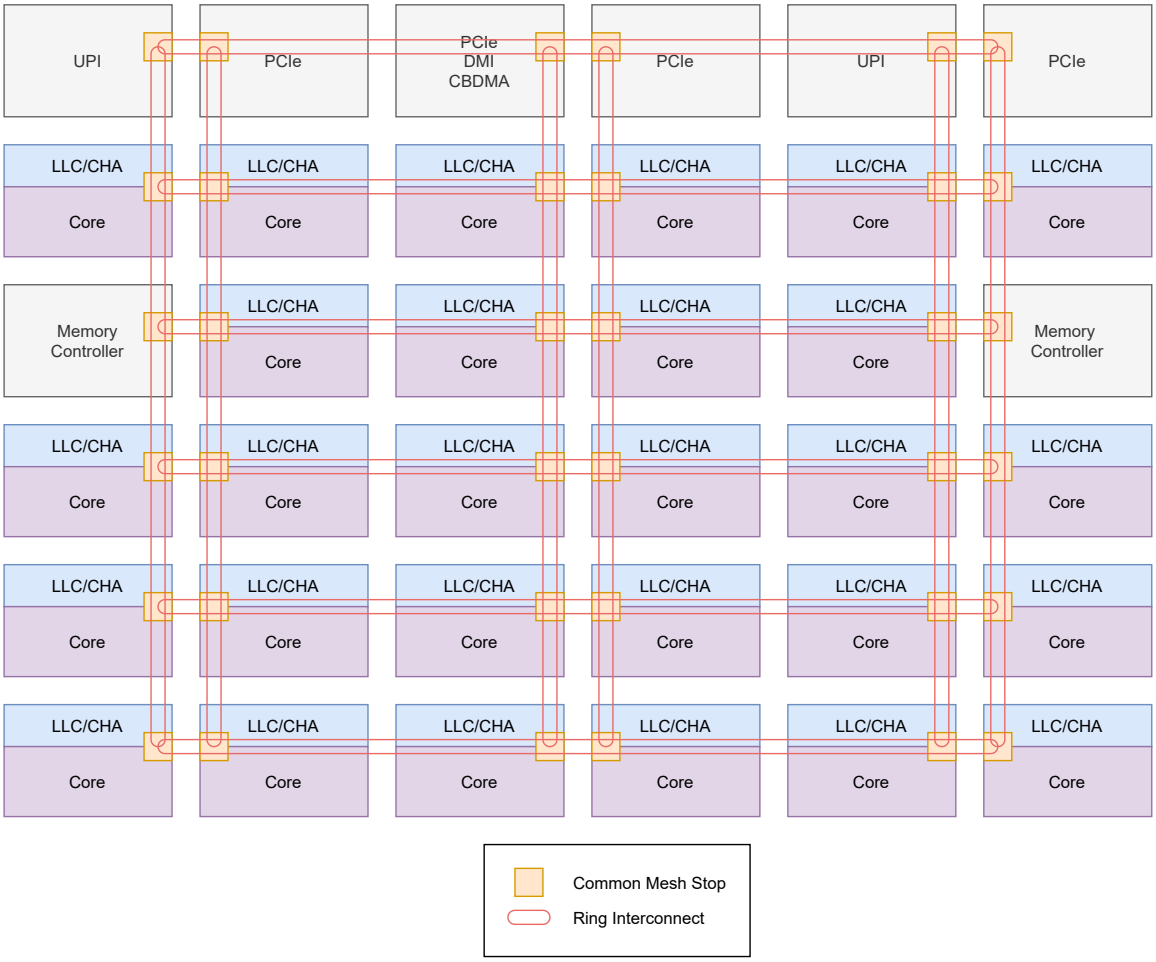

Figure 2-2: Intel mesh interconnect on a 28-Core die configuration.

Tile Components

The tile itself plays a large role in the routing of the packets. In addition to the components specific to each tile's function, all tiles on the mesh interconnect contain a *Common Mesh Stop (CMS)* as shown in Figure 2-2. The CMS is a common interface

to the mesh that can be shared by all tiles, providing modularity to the interconnect. The role of the CMS is to inject, receive, and forward packets on the mesh. A large component of our reverse-engineering efforts focus on the behavior of the CMS.

While there is no official documentation of the CMS structure, a patent filed by Intel bears a strong resemblance to Intel's documentation of the Cascade Lake Uncore [23]. The patent for an *Anti-Starvation and Bounce-Reduction Mechanism for a Two-Dimentional Bufferless Interconnect* describes each CMS as sitting at the intersection between the horizontal rings and the vertical rings. The CMS itself contains a horizontal ring stop and a vertical ring stop (Figure 2-3).

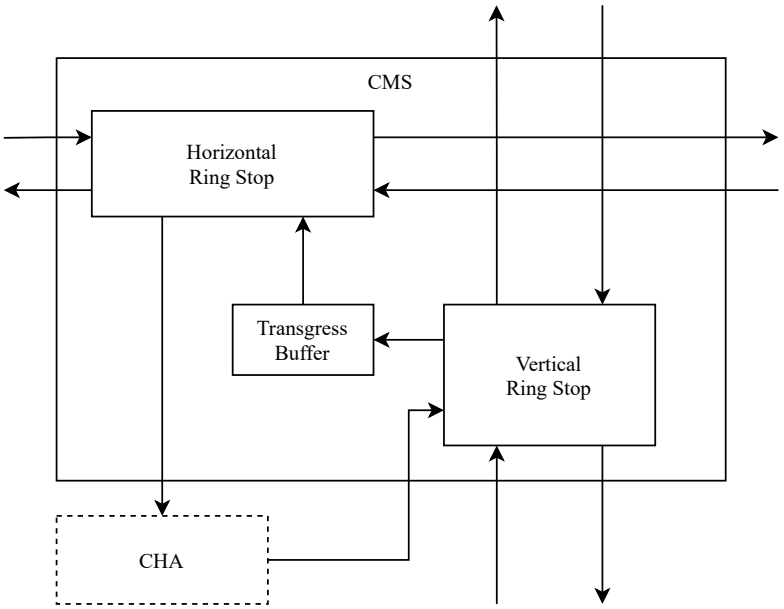

Figure 2-3: The structure of a Common Mesh Stop

In the Y-X routing scheme, it may be necessary for a packet to switch from the vertical to the horizontal ring. This is accomplished through the *transgress buffer*. A packet traversing the interconnect from a local core to a remote core will first exit the CHA and be injected into the vertical rings via the vertical ring stop. Upon reaching the correct horizontal ring, the packet will enter the transgress buffer where it will wait for an empty slot to be injected into the horizontal ring stop. The packet then traverses horizontally until it reaches its target tile, at which point it will enter the

target tile's CHA and be consumed by the core.[2] Importantly, the patent also claims that at each interface between the CMS and the ring, packets currently in flight in the ring will take precedence over packets that are waiting to be injected into the ring [23], a fact that we confirm in Chapter 4.

2.3 Microarchitectural Side Channels

We end this chapter with a brief overview of how shared hardware on multi-core processors can lead to security issues through microarchitectural side channels. In general, a microarchitectural side channel involves a *transmitter* in the victim's security domain and a *receiver* in the attacker's security domain stealthily communicating with each other. The medium of the communication channel is some microarchitectural structure(s) that is shared by the transmitter and the receiver, thus allowing their states and occupancy to be modified and observed by the transmitter and the receiver's activities. We classify microarchitectural side channels into two categories based on the type of resource they are exploiting: state-based attacks and scheduler-based attacks.

State-based attacks focus on microarchitectural resources that hold shared states, such as caches [8], DRAM row buffers [26] and Branch Target Buffers (BTBs) [12, 13]. In particular, caches have been extensively studied, and prior work has shown that the presence states [25], replacement states [16, 31] and coherence states [33] can all be exploited to construct side channels. A state-based attack generally involves three steps. First, the receiver executes and brings a shared microarchitectural structure into a known state. Second, the transmitter is triggered to modify the shared state based on some secret value. Third, the receiver probes the structure to learn the modified state and infer the secret. The classic cache attacks, such as Flush+Reload [34] and Prime+Probe [25], follow the three steps above.

Scheduler-based attacks focus on exploiting the *occupancy* of a resource that is

[2]This description is actually the transpose of the mechanism described in the patent [23]. The example given in the patent uses X-Y routing, so all horizontal and vertical directions have been flipped in this discussion to maintain consistency with the Y-X routing on our processor.

shared by multiple programs where the occupancy is mediated by a scheduler. Many hardware resources, including functional units [29], cache banks [36], and memory buses [30], are associated with a scheduler. When multiple requests for service arrive at such a resource, the associated scheduler needs to decide the order in which these requests will use the resource. As a result, the scheduler introduces extra latency to some of the requests. If the resource is shared by an attacker and a victim and that extra latency is observable by the attacker, it might result in information leaks. Note that, in a scheduler-based attack, information leakage happens only *during the time* when the victim is occupying the resource, which is very different from a state-based attack, where the attacker and the victim does not need to simultaneously use the targeted resource. Attacks on the on-chip network [24] typically fall in this category. The network is the shared resource, and, on our processor, the CMS performs the scheduling. When packets from the victim are in flight on the network, packets from the attacker may experience delays due to contention for bandwidth on the interconnect. In this way, the network itself may be used as a microarchitectural side channel.

Chapter 3

Reverse Engineering the Interconnect Layout and Scheduling Policy

In this chapter, we introduce Intel's Uncore Performance Monitoring system and demonstrate how it can be used to determine the positions of the cores. We then continue using performance monitors to investigate how cache coherence traffic flows through the interconnect.

3.1 Uncore Performance Counters

The Intel "Uncore" refers to logic that resides outside of the CPU cores but within the same die [6]. This includes data reads, cache coherence traffic, and power management messages. For our reverse-engineering efforts, we are interested in monitoring the *caching and homing agent (CHA)* that is co-located with the LLC on each tile and the *integrated memory controller (IMC)* that is located on the left and right edge of the die (Figure 2-2).

3.1.1 Performance Counter Organization

The Uncore performance monitors on Cascade Lake processors are managed in a two-level hierarchy: a number of performance monitoring (PMON) blocks can be

individually configured and are all governed by a PMON Global Block. Each block contains a pair of *Unit Control and Status Registers* that allow for configuration and monitoring of the entire PMON block. In addition, each PMON block contains four pairs of *Counter Registers* and *Counter Control Registers* that can be individually configured to count specific events and even filter for subsets of those events through the use of *unit masks* or *umasks*. The unit control registers are used to freeze and reset the entire PMON block. The status registers indicate which (if any) of the four counters within the block experienced a counter overflow event. The PMON Global Block is able to simultaneously start and stop all PMON blocks as well as detect overflow events. All control registers, status registers, and counters are implemented as model-specific registers (MSRs). They can be accessed using the Linux MSR kernel module which exposes them at `/dev/cpu/*/msr` [2].

There are multiple types of hardware modules within the Uncore that have attached performance monitors. These are referred to as *boxes*. For example, there are up to 28 CHA boxes, up to 2 IMC boxes, and up to 2 UPI boxes. Each box is able to measure a set of events that is relevant to its function. The CHA boxes, for example, can measure the number of LLC lookups that have occurred. There are also mesh performance monitoring events (CMS events) that are accessible from CHA, M2M, and M3UPI boxes.

3.1.2 Counters of Interest

There are many uncore performance counter events listed in Intel's Uncore Performance Monitoring manual [6]. Below, we summarize a few useful ones for our reverse-engineering experiments.

CMS Clockticks

The `CMS_CLOCKTICKS` event is helpful for debugging and ensuring that the counters are being started, stopped, and reset properly. The clockticks counter will count the number of CMS cycles that have elapsed during the performance monitoring session.

Ring Usage

There are a set of 8 performance counter events that measure the number of clock cycles during which each ring of a particular direction is being used at the CMS. They are identified by a direction and a ring (e.g. `HORZ_RING_BL_IN_USE`, `VERT_RING_AD_IN_USE`). Each one monitors one of the four rings in either the vertical or horizontal direction. To use these events, identify the direction to monitor, and set the proper umask. Note that umasks can be bitwise OR'ed together to count multiple subcategories of each event. There are three important details to observe when using these performance counters.

- In the performance monitoring manual, there is a reference to "even" and "odd" rings in the listing of the umasks [6]. While not explained in the manual, we believe this refers to the Lanes discussed in Section 3.4. To get the total count of the packets traveling in a particular direction, we can bitwise OR the umasks for the even and odd lanes for that direction to count both lanes.

- These performance counters only track packets in flight or being sunk by the current CMS. They do not track packets that originate from the core or the LLC slice attached to the current CMS.

- In every other column, the direction of the horizontal performance monitors is reversed. This is due to the layout of the cores on the die. Observe from Figure 2-2 that the cores on every other column are mirrored horizontally. This means that configuring a counter to measure packets on the horizontal ring going to the left actually measures packets on the horizontal ring going to the right. This observation was first documented in John McCalpin's work on core mapping [21].

LLC Lookups

The `LLC_LOOKUP` event is specific to CHA boxes and measures the number of times the LLC was accessed. This event counts every access to the LLC regardless of purpose,

including code, data, prefetches, and hints from L2. To use this event effectively, it is important to filter with the umasks. For our purposes, we will use this event in conjunction with the `clflush` command, so we are aiming to count data reads. This suggests we set the umask to `DATA_READ`.

3.2 Target Architecture and Tile Layout

Before reverse engineering the network traffic patterns, it is necessary to understand the detailed layout of the processor tiles on the die. In this section, we describe key architectural parameters of the Intel Xeon Scalable Family Processors on the Purley platform launched in 2017, which uses the Cascade Lake architecture and implements the mesh interconnect. These parameters include the cache configurations and the tile layout.

3.2.1 Processor Details

The experiments in this work were conducted on a two-socket motherboard containing two Intel Xeon Gold 5220R (Cascade Lake) CPUs at 2.20 GHz. Note that Cascade Lake is an optimization of Skylake. Most of the cache and tile configurations discussed in this section also apply to Intel Skylake server microarchitectures. We limit the bulk of our experiments to the first socket using a combination of operating system CPU pinning [5] and sub-NUMA clustering [3] to keep the memory allocations to the on-chip memory controllers since this work focuses on studying the on-chip network rather than the inter-chip connections such as the UPI.

3.2.2 Cache Configurations

The Intel Cascade Lake server microarchitecture contains three levels of caches shown in Table 3.1. The capacity of each cache and its associativity may be obtained from the Intel 64 and IA-32 Architectures Optimization Reference [7]. The number of sets at each level of cache can be computed from the associativity and the block size (64

bytes). For example, for L1:

$$\text{L1 sets} = \frac{32\,\text{KiB}}{64\,\text{B/line} \times 8\,\text{lines/set}} = 64\,\text{sets}$$

Memory Structure	Capacity	Associativity	Sets
L1	32 kB	8	64
L2	1 MB	16	1024
LLC (non-inclusive)	1.375 MB/slice	11	2048
Extended Directory		12	2048

Table 3.1: Cache configurations of Intel Cascade Lake server microarchitecture.

The total LLC capacity is dependent on the machine configuration. The value provided in the Intel reference manual states 1.375 MB/slice. In Section 3.2.3, we will calculate the number of slices on our processor. Also of note is the fact that the LLC is non-inclusive [7]. The 12-way associativity of the directory was confirmed by the methods described in [32]. These two facts combine to present some complications in the eviction and monitoring strategies in the design of the receiver as described in Section 4.1.

3.2.3 Tile Layout

Intel Cascade Lake processors come with three different die configurations, namely, LCC (low core count), HCC (high core count), XCC (extreme core count). We focus our analysis on the XCC configuration, which consists of 30 tiles, organized into a 5×6 grid, shown in Figure 3-1.

On XCC dies, certain cores may be partially or fully disabled. To reverse engineer the core layout and identify the disabled tiles, we begin by looking at some basic machine statistics with `lscpu` in Listing 3.1 and taking stock of the active tile counts.

It is known from publicly-available documentation that two tiles are IMC tiles. The remaining 28 tiles could potentially be core tiles. Line 6 in Listing 3.1 shows 24 cores on each socket which suggests that four core tiles must be disabled.[1] Further,

[1]Prior work [21] and our observations confirm that while disabled cores have CMS modules and

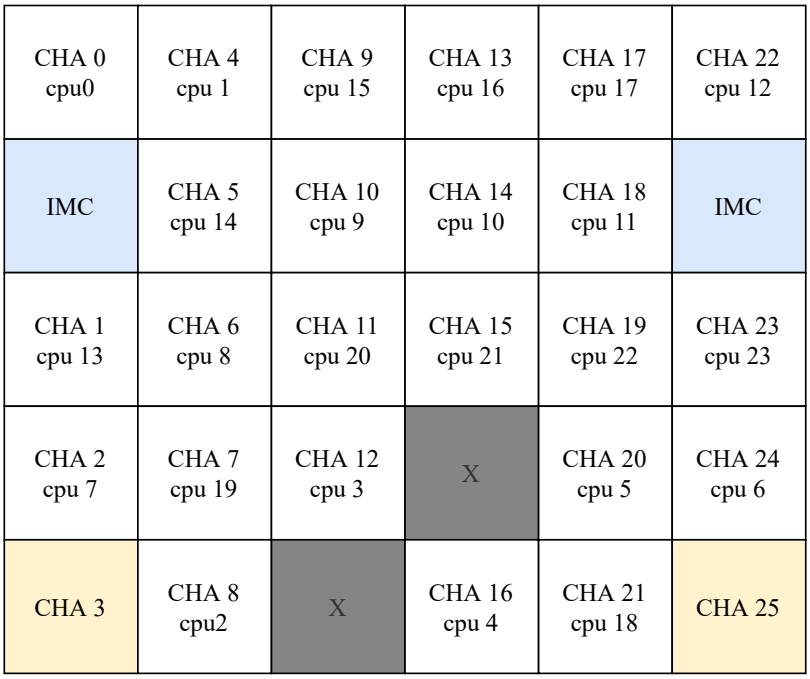

Figure 3-1: An example tile layout of an Intel Cascade Lake processor with 24 active cores and 26 active LLC slices.

dividing the L3 cache size (`36608K` shown in Line 13) by the LLC per-slice capacity (1.375 MB) yields a total of 26 LLC slices. This suggests that of the four disabled core tiles, two still have active LLC slices. The next task is to determine their locations.

```
1  $ lscpu
2  Architecture:        x86_64
3  CPU(s):              96
4  On-line CPU(s) list: 0-95
5  Thread(s) per core:  2
6  Core(s) per socket:  24
7  Socket(s):           2
8  NUMA node(s):        4
9  Model name:          Intel(R) Xeon(R) Gold 5220R CPU @ 2.20GHz
10 L1d cache:           32K
11 L1i cache:           32K
12 L2 cache:            1024K
13 L3 cache:            36608K
14 ...
```

Listing 3.1: Selected output of `lscpu`.

do connect to the mesh to forward packets, they do not contain CHA slices and do not contain performance counters.

Searching for Disabled Tiles

The disabled tiles can be queried directly through the CAPID6 capability register as described in the Intel Xeon Uncore Performance Monitoring manual [6]. This register returns a bitmap that indicates the position of the available CHAs. CAPID6 can be accessed as a PCI device at Device 30 (0x1e), Function 3, Offset 0x9C [6]. We can use `lscpi` to find the exact address first and then query the register with `setpci` as shown in Listing 3.2.

```
1 $ lspci | grep :1e.3
2 17:1e.3 System peripheral: Intel Corporation Sky Lake-E PCU
     Registers (rev 07)
3 85:1e.3 System peripheral: Intel Corporation Sky Lake-E PCU
     Registers (rev 07)
4 $ sudo setpci -s 17:1e.3 0x9c.l
5 0ffddfff
6 $ sudo setpci -s 85:1e.3 0x9c.l
7 07fff7ff
```

Listing 3.2: Reading CAPID6 in the PCI space.

The two values returned correspond to the CAPID6 registers for the two sockets on our machine. The value `0ffddfff` is a 28-bit number that corresponds to the 28 core tiles on the die, counting from the top left and going in column-major order. There are zeros at bits 13 and 17, indicating that core tiles 13 and 17 are disabled. The location of the disabled tiles is not consistent between different dies. From Listing 3.2, it is evident that the die in the second socket contains disabled cores at different locations. The remaining CHAs and LLC slices are numbered in column-major order while skipping over the disabled cores as shown in Figure 3-1. We will refer to this numbering as the *CHA ID* and will primarily use this ID to refer to specific tiles unless otherwise specified.

CPU to CHA Mapping

With the knowledge about the locations of the two disabled cores, the last remaining task is to determine how to interface with active cores from the Linux userspace. Linux exposes 96 CPUs as shown in Listing 3.1. CPUs 0-23 and 48-71 correspond to Socket 0. CPUs 24-47 and 72-95 correspond to Socket 1. Within each socket, CPUs n

and $n+48$ form a pair of Intel Hyper-threads, Intel's implementation of simultaneous multi-threading. Therefore, in terms of physical hardware, there are 24 cores on each socket. Our goal is to figure out which Linux CPU ID corresponds to which physical tile.

There is a substantial amount of prior work that demonstrates various approaches to determining this mapping. We followed McCalpin's approach [22] using performance counters. McCalpin's insight was to recognize that if we pin a process to a particular CPU and then repeatedly access DRAM, we should observe traffic flows from both IMCs. Due to the Y-X routing of the processor, only the process' pinned tile should see traffic entering from two directions.

The test program we use first pins itself to a core using the CPU ID with `sched_setaffinity` [5] and allocates a large segment of memory to encourage traffic from both IMCs. It can then force an access to the DRAM using the `clflush` instruction to flush an address from all levels of cache before reloading it, requiring data to flow from the IMC back to the pinned core along the block ring [1]. By monitoring the `HORZ_RING_BL_IN_USE` and `VERT_RING_BL_IN_USE` performance counters (which are indexed using CHA IDs) on each core, it is possible to determine which core with a particular CHA ID is receiving traffic from two directions. This provides the mapping between the Linux CPU ID used to pin the process initially and the CHA ID. The results of this reverse-engineering are summarized in Figure 3-1.

3.3 Traffic Flows

With the positions of the cores determined, we now seek to understand the traffic flows involved in implementing the cache coherence protocol.

3.3.1 Experiment Setup

To generate traffic on the interconnect, we run a process we will call the *transmitter*. The goal of the transmitter is to create traffic between a core and an LLC slice. The transmitter will pin itself to the desired core and create two sets of addresses known

as *eviction sets (EVs)*. The addresses within an EV all map to the same L2 set and the same LLC slice. Note that because the LLC has twice as many sets as the L2, the addresses in the EV will map to two different LLC sets. We configure the two EVs in the transmitter to map to the same L2 set but to different LLC slices: one EV (the *Local EV*) maps to the slice co-located with the transmitter's core while the other EV (the *Remote EV*) maps to the desired LLC slice that the transmitter is sending traffic to. The EVs each contain 20 addresses. When we access the addresses in the Local EV, it evicts the addresses of the Remote EV from L2. Now, when we access the addresses of the Remote EV, they must be fetched from the LLC, producing interconnect traffic to the desired target slice.

Algorithms for EV generation have been described by prior work [20]. Here, we discuss two practical concerns associated with generating these address sets.

Finding Set Indices

The key practical challenge in finding the set index on a particular address comes from the virtual memory system. The address translation process prevents the program from seeing the physical address, making it impossible extract the set index bits. However, we can take advantage of the fact that during an address translation, the page offset remains constant – only the virtual page number is replaced by the physical page number. For example, in an operating system using a 4 kB page size (2^{12} bits), the lower 12 bits remain constant during the address translation. On a typical Linux system, it is possible to enable pages as large as 2 MB and 1 GB. With 2 MB pages, the lower 21 bits remain constant, allowing the user-level process to figure out which cache set a virtual address belongs to without accessing the physical address. This still leaves the question of determining the LLC slice through the slice hash function.

Finding Slice Mappings

As mentioned in Chapter 2, the slice hash function that maps addresses to LLC slices is undocumented in Intel processors. We can circumvent this with performance monitors. To find the mapping of an address A to an LLC slice, we first configure the

LLC_LOOKUP event on all the CHA PMON boxes with the DATA_READ umask. Next, we repeatedly access and flush address A from the cache with clflush. This will require performing data reads in the LLC. The access will miss in the LLC, but the read request will still be counted. Lastly, we compare all the counts across the CHAs and one should be significantly higher than the rest. That indicates the CHA slice that is being accessed and that address A maps to.

3.3.2 Observing Traffic Flows

To understand the kinds of traffic flows produced by an LLC access, we augment an instance of the transmitter with hardware performance monitors to observe the types of packets sent and the paths they take. The transmitter is set to send traffic across the top row of the processor from CHA 0 to CHA 22 by pinning itself to the CPU corresponding to CHA 0 (CPU 0) and using a remote EV that maps to CHA 22. We indicate this configuration as follows: Core 0 $\leftrightarrow$ Slice 22.

By setting up the four counters within each PMON box to measure the up, down, left, and right directions for one ring at a time, we can observe how traffic from various rings flows through the network. For example, to measure traffic on the block (BL) ring, we set the four counters in each PMON box to the configuration in Table 3.2. Recall from Section 3.1 that these counters will increment when a packet is present on the BL ring in a particular direction at a certain CMS. It measures packets that are in flight and also packets that are sunk by the tile but not packets sourced by the tile.

Event	umask	Description
VERT_RING_BL_IN_USE	0x3	Up ring
VERT_RING_BL_IN_USE	0xC	Down ring
HORZ_RING_BL_IN_USE	0x3	Left ring
HORZ_RING_BL_IN_USE	0xC	Right ring

Table 3.2: Configuration of PMON box counters for measuring Block Ring traffic in four directions.

Running the transmitter above produces the output summarized in Table 3.3.

Bearing in mind that every other column has an inverted horizontal counter, we can derive the flows shown in Figure 3-2. Note that performance counters track packets *arriving at* the the CMS, so, for example, the arrow from CHA 4 to CHA 9 corresponds to the Right counter for CHA 9.

CHA ID	Up	Down	Left	Right
0	0	0	8003	0
4	0	0	8002	8010
9	0	0	8004	8001
13	0	0	8000	8002
17	0	0	8000	8018
22	0	0	8010	0

Table 3.3: Block Ring performance monitoring counts (scaled down by 1000x) with negligibly small values omitted.

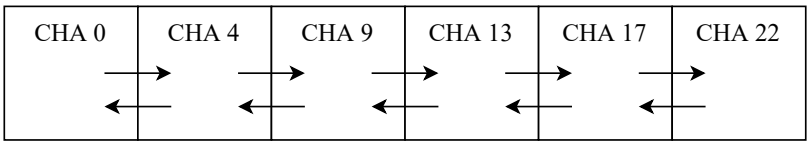

Figure 3-2: Block Ring flows when the transmitter sends from Core 0 to Slice 22.

Repeating this experiment for all four rings reveals the traffic flows in Table 3.4. We observe that the transmitter generates traffic from the core to the LLC slice on the address ring and the block ring. In the other direction, there is also traffic from the LLC slice to the core on the block ring. This suggests the following steps of the cache coherence protocol when accessing an address from the LLC:

- Core sends a request on AD to the slice.
- Slice sends data on BL to core.
- Core writes back the data from its private cache to the slice on BL.

Due to the non-inclusive nature of the LLC, an L2 eviction requires that the line be written back to the LLC. This is different from an inclusive cache in which a line present in the L2 must also be present in the LLC. A simple invalidation would be

Ring	Core→Slice	Slice→Core
Address (AD)	✓	
Block (BL)	✓	✓
Acknowledge (AK)		✓
Invalidate (IV)		

Table 3.4: Traffic flows when accessing a LLC slice.

sufficient to evict the line from L2. In a non-inclusive cache, the line may not be present in the LLC despite being present in the L2 so eviction from L2 requires a writeback operation.

3.4 Multi-Lane Interconnect

Prior work by Paccagnella et al. [24] on reverse-engineering the Intel ring interconnect revealed that each ring was actually implemented by a pair of rings, which we will refer to as *lanes*. This is hinted at in the performance counter documentation [6]. In the description of the umasks for CMS performance monitoring, there is a mention of "Even" and "Odd" rings. As we were not able to find a conclusive explanation for the parity-related terms, we will refer to the lanes as A and B in our analysis.

Applying finer-grained filtering via the umasks allows us to see how traffic is being allocated onto lanes. We configure the transmitter to send horizontally and set the performance monitors to count traffic on the Even and Odd lanes (Table 3.5).

Event	**umask**	**Description**
`HORZ_RING_BL_IN_USE`	0x1	Left ring, Even
`HORZ_RING_BL_IN_USE`	0x2	Left ring, Odd
`HORZ_RING_BL_IN_USE`	0x4	Right ring, Even
`HORZ_RING_BL_IN_USE`	0x8	Right ring, Odd

Table 3.5: Configuration of PMON box counters for measuring horizontal Block Ring lane traffic.

We vary the transmitter configuration and record which lane is used by the traffic. This yields the *lane schedule policy* for Row 0 shown in Figure 3-3 (a). Note that traffic from Slice→Core uses the opposite lane as the traffic from Core→Slice.

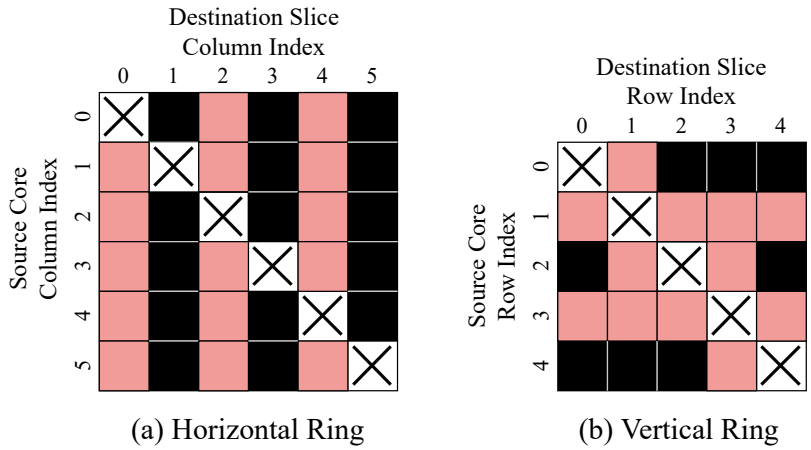

Figure 3-3: Lane schedule policy for Core→Slice traffic on horizontal rings and vertical rings. Black and red squares indicate different lanes. For each pair of source and destination tiles, the Slice→Core traffic uses the opposite lane.

These results allow us to infer the structure of the lane organization in Row 0. The vertical pattern in Figure 3-3 (a) suggests that the choice of lane is determined by the destination. To explain this, observe that any core and slice can inject packets onto both lanes, but not every core or slice can receive packets from both lanes. Specifically, on a horizontal ring on Row 0 of the network, the egress ports of Lane A connect to the cores on even columns and the slices on odd columns, and the egress ports of Lane B connect to the cores on the odd columns and the slices on the even columns, as shown in 3-4. Because the packet cannot arbitrarily exit the network on any lane, the selected lane for a given packet must be determined by the destination of the packet.

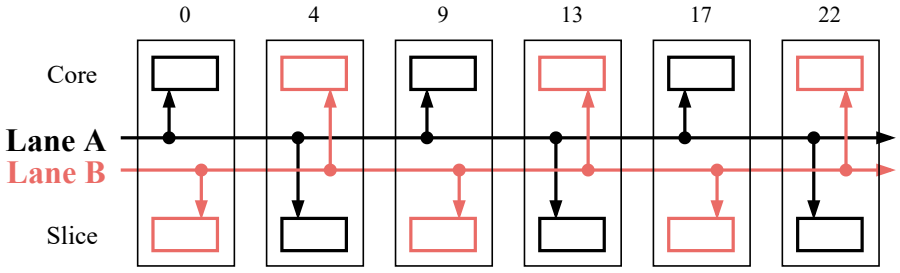

Figure 3-4: An example multi-lane organization for a ring on Row 0.

We repeat this experiment vertically to obtain the vertical lane scheduling policy shown in Figure 3-3 (b). One unexpected behavior we observed was that the

performance counter results for the vertical lane showed packets appearing to switch between the even odd lanes on each hop. We believe this is a similar effect to how the horizontal counters in every other column were reversed. We flipped every other row in our vertical-lane experiments to arrive at the results above. In contrast to the horizontal scheduling policy, the vertical scheduling policy appears to follow a source-based policy with four special cases where the lane selection is inverted: $0 \rightarrow 1, 2 \rightarrow 1, 2 \rightarrow 3$, and $4 \rightarrow 3$. We believe these cases were adjusted to help load balance the network since there are an odd number of cores and slices on the vertical ring. Further, we believe that a source-based policy on the vertical ring is consistent with our understanding of the CMS. Packets always follow a Y-X route which involves first traveling vertically until encountering the CMS on the correct row. Then the packet must enter the transgress buffer before switching to the horizontal lane. Because the packet never travels directly from the vertical lane to the core or slice, there is no reason to partition the lanes by the destination as seen on the horizontal policy.

Chapter 4

Timing Analysis of Interconnect Contention

A prerequisite for covert and side channel attacks via the mesh interconnect is an understanding of how various traffic flows contend for bandwidth on the network. Based on our reverse engineering results from Chapter 3, we can now perform an in-depth timing analysis of interconnect contention. Specifically, we use a pair of processes: a *transmitter* and a *receiver* to create bandwidth contention. The receiver times a set of carefully controlled memory accesses that require traversing the interconnect. The transmitter produces traffic that will interfere with that of the receiver, causing delays in the timed accesses.

4.1 Designing the Receiver

In this section, we describe the design of the receiver process and how it monitors the traffic on the interconnect. The goal of the receiver is to time a series of memory loads that require traversing the interconnect. Increases in the access latency indicate the presence of contention from other traffic flows along the path taken by the data packets.

At a high level, the receiver will pin itself onto a pre-determined core R_C and generate a set of addresses, known as the *monitoring set (MS)* that all map to the

desired target LLC slice R_S. The receiver then sets up the addresses to miss in the L1 and L2 caches and hit in the LLC, forcing traffic on the interconnect. These loads can be timed and any contention should cause a latency increase.

Designing the memory accesses requires an understanding of the cache structures on the processor. Specifically, the high associativity of the L2 cache and the non-inclusivity of the cache present challenges to ensuring that the memory accesses are all served from the LLC. Prior work [24] introduced a strategy that constructs a monitoring set of size W_{LLC}, the associativity of the LLC. Taking advantage of the fact that the associativity of their LLC is greater than that of both L1 and L2 ($W_{LLC} > W_{L1}, W_{L2}$) on their processor, Paccagnella et. al. [24] were able to sequentially access all W_{LLC} addresses, and because all addresses could not fit within the private caches, this access pattern would necessarily cause LLC accesses. However, on our Cascade Lake processor, $W_{LLC} < W_{L2}$ since our L2 cache has 16 ways while our LLC only has 11 ways. To solve this problem, we introduce an *eviction set (EV)*, a set of addresses designed to conflict with the monitoring set in the L2 but not the LLC. Our receiver uses the following procedure to produce reliable LLC hits:

1. Set up eviction and monitoring sets.

    (a) Generate a monitoring set of size W_{MS} where $W_{MS} \leq 2 \times W_{LLC}$. The addresses within the monitoring set map to the same L2 set and the same LLC slice, R_S.

    (b) Generate an eviction set of size W_{EV} where $W_{EV} \geq W_{L2}$. These addresses map to the same L2 set as the monitoring set but to a different LLC slice from R_S, specifically, to the LLC slice co-located with R_C.

    (c) Access all addresses within the monitoring set. This loads all monitoring-set addresses into the private cache on R_C. However, due to the non-inclusive nature of the cache, these addresses are not necessarily present in the LLC.

2. Perform latency measurements.

(a) Access the addresses of the eviction set multiple times. This will evict any monitoring-set addresses from the L2. Additionally, any monitoring-set addresses not already present in the LLC will now be written back from the L2 into the LLC. At the end of this step, all addresses of the monitoring set will reside in the LLC.

(b) Sequentially load the addresses in the monitoring set, and time the latency of this operation.

Step 1 is performed once on receiver initialization. Step 2 is performed repeatedly to collected multiple latency measurements.

4.1.1 Monitoring Set Design

The monitoring set has several parameters that can be tuned to increase the efficiency and sensitivity of the receiver.

Load Timing

In order to achieve reliable timing measurements, it is important to control the nature of the memory accesses being timed. Modern processors perform out-of-order execution which allows the processor to reorder instructions and execute them in parallel if it improves throughput. This introduces noise into our latency measurements as parts of our monitoring set may be accessed in parallel which causes inconsistent latency measurements. To combat this, we force the processor to serialize these loads using a technique known as *pointer-chasing*, first introduced by Tromer et al. [27]. In this setup, the memory at each address in the MS holds the address of the subsequent element. The last element of the set points back to the first, forming a circularly-linked list. As shown in Listing 4.1, the MS is accessed by successively dereferencing pointers to traverse the linked list. Because accessing the next MS address depends on the address obtained by the current access, these loads cannot be parallelized. Two calls to the `rdtscp` instruction can be used to read the timestamp counter and time the loads. Once again, it is important to use `rdtscp` instead of `rdtsc` to ensure that

```
1  void **current = monitoring_set;
2  asm volatile(
3      ".align 32\n\t"
4      "lfence\n\t"
5      "rdtsc\n\t"             // eax = TSC (timestamp counter)
6      "movl %%eax, %%r8d\n\t" // r8d = eax; store current TSC
7
8      "movq (%1), %1\n\t"     // current = *current
9      "movq (%1), %1\n\t"     // current = *current
10     ...
11     "movq (%1), %1\n\t"     // current = *current
12
13     "rdtscp\n\t"            // eax = TSC (timestamp counter)
14     "sub %%r8d, %%eax\n\t"  // eax = eax - r8d; get elapsed time
15     "movl %%eax, %0\n\t"    // timings[i] = eax
16
17     : "=rm"(timings[i]), "+rm"(current) // output
18     :
19     : "rax", "rcx", "rdx", "r8", "memory");
```

Listing 4.1: Timing a single access of the monitoring set

the timestamp counter is read only after the loads are performed [4]. Careful uses of the `lfence` instruction further help enforce the instruction ordering constraints.

Monitoring Set Size

The size of the monitoring set (MS) plays a key role in the efficiency-sensitivity trade-off. A larger MS requires more time to iterate over but can also amplify the contention signal observed. For our reverse-engineering experiments, we wish to maximize the dynamic range of our receiver to improve the chances of distinguishing different levels of contention. To this end, we examine how large a monitoring set can be in theory and consider what works in practice.

The maximum allowable size of the MS is determined by the cache structure and some implementation-specific details. While a simple understanding of the cache suggests that the MS can be as large as the associativity of the L2 (16 for the Cascade Lake architecture), the MS size is also upper-bounded by the directory structure used by Intel to implement the distributed coherence protocol for the non-inclusive LLC. Prior work has demonstrated that the directory has an associativity of 12 [32]. This can be confirmed by running the the receiver without evictions and observing at which

point we begin to see DRAM accesses. We observe that after a monitoring-set size of 12, the average access latency increases into the range of DRAM accesses. We can circumvent this restriction by allowing the MS to reside in two LLC sets instead of just one.[1] This also distributes the directory entries between two sets. This allows us to achieve a MS size of up to 24 without violating the directory constraint. Combining the two constraints of the L2 associativity and the directory associativity, we arrive at a maximum MS size of 16.

In practice, we found that a MS size of 16 allows for high sensitivity to the transmitter but proved to be unreliable across many runs and in many configurations due to the relatively high likelihood of spurious MS evictions. Instead, we found that a MS size of 4 was enough to observe contention and proved to be sufficiently reliable. When high sensitivities are needed, however, a larger MS size is a useful knob to adjust.

4.1.2 Eviction Set Design

The role of the eviction set (EV) is to reliably place the monitoring set exclusively within the LLC. Because the eviction set accesses do not directly contribute to the sample collection, there does not need to be as much care taken in the memory accesses. The loads to the EV addresses can be executed in parallel without pointer-chasing.

The effectiveness of the eviction is highly dependent on the cache replacement policy, the exact details of which are proprietary and likely complex [28]. To ensure eviction of the monitoring set, we access the EV multiple times. Our experimentation found that at least two rounds of accesses were required to successfully evict the MS and that excessive accesses can cause unpredictable results. We selected four rounds of EV accesses per each access of the MS to provide the highest reliability for our reverse-engineering work. To achieve higher efficiency in later experiments, it is possible to

[1]The LLC has twice as many sets as the L2, so an address that maps to an L2 set can map to one of two LLC sets. This allows the MS to reside within the same L2 set while being distributed between two LLC sets.

reduce this value.

The last aspect of the EV to consider is the pattern in which the EV addresses are accessed. Prior work in eviction sets recommends testing a number of access patterns [14]. We tested three of the most common patterns:

Linear

$EV[0], EV[1], EV[2], EV[3], ...$

ABABAB

$EV[0], EV[1], EV[0], EV[1], EV[0], EV[1], EV[1], EV[2], EV[1], EV[2], ...$

ABCABC

$EV[0], EV[1], EV[2], EV[0], EV[1], EV[2], EV[1], EV[2], EV[3], EV[1], ...$

Empirically, we found that the ABCABC pattern produced the best results.

4.2 Timing Characteristics of the Mesh Interconnect

We ran the receiver on different Core↔Slice configurations, collecting 100,000 samples on each configuration. We observed that, regardless of the configuration and the system load, the resulting trace was bimodal. We can clustered the samples into two *cohorts*, which we call the *fast cohort* and the *slow cohort*. Figure 4-1 shows an example of these two cohorts when the receiver has configuration Core 13 ↔ Slice 9. Note that the fast cohort is the bottom cluster (fewer cycles per sample), and the slow cohort is the top cluster. In this example, there is a difference of 10 cycles between the fast and slow cohorts. The samples aperiodically scale up or down between these two cohorts. After investigating and attempting to control more factors on the machine, we do not have an explanation for the presence of the cohorts, but we suspect that given its longer time-scale on which the switching occurs, it may be due to frequency scaling on the CPU interconnect.

Recall that our experimental methodology requires the ability to compare traces in the presence or absence of a transmitter. Prior work achieved this by computing

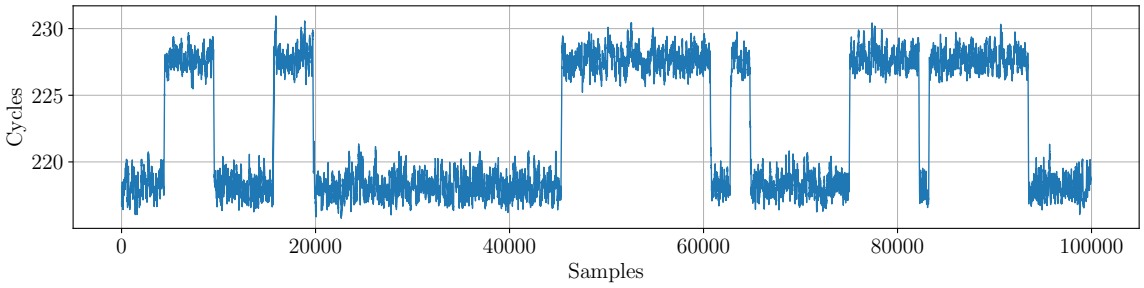

Figure 4-1: Example receiver trace.

the average and standard deviation of each 100,000-sample trace [24]. However, the irregular frequency scaling we observed on our CPU makes these metrics unreliable and too coarse-grained. We deal with this challenge by classifying the samples into two sets, one for each cohort. Because the cycle times required change based on the length of the path of the receiver, it is not feasible to simply predefine a threshold since it would need to be redefined for each receiver configuration and would not be reliable when processing a trace from a run with contention. Instead, we rely on a K-means classifer with $k = 2$ to automatically perform the split.

Finally, we compute the average and standard deviation of the two levels separately. We empirically verified that this approach reliably separates the two levels and produces more accurate averages and standard deviations. In the rest of this work, for simplicity, we will compare traces using the metrics computed on the fast cohort only.

4.3 Designing the Transmitter

To generate traffic on the interconnect, we optimize the transmitter described in Section 3.3. The transmitter's purpose is to generate traffic on the interconnect to contend with the monitoring of the receiver. Ideally, the transmitter should produce a large amount of consistent traffic. Furthermore, it should do so with a minimum amount of additional interconnect traffic to avoid creating false positives on the receiver.

Recall that the transmitter accesses two eviction sets (EVs) in sequence: the local

EV and the remote EV. The two eviction sets mutually evict each other and, in the process, generate traffic from the core to the target LLC slice. In contrast to the receiver, the accesses to the address within each EV are performed in parallel without the pointer-chasing setup, allowing for faster bursts of interconnect traffic.

4.3.1 Transmitter Tuning

The basic transmitter can be tuned to produce higher amounts of contention and increase the signal observed by the receiver. To evaluate the transmitter, we use the receiver described in Section 4.1. The receiver monitors the link between Core 13 $\leftrightarrow$ Slice 9. The transmitter operates on Core 22 $\leftrightarrow$ Slice 4. To evaluate the efficacy of the transmitter, we perform two runs: one with the transmitter on and another with the transmitter off. The latencies obtained by the receiver in both cases are then subtracted to determine the amount of contention caused by the transmitter. Figure 4-2 shows an example of the trace observed by the receiver with the transmitter off and on. Contention causes the transmitter-on trace to slower (higher latency) than the transmitter-off trace.

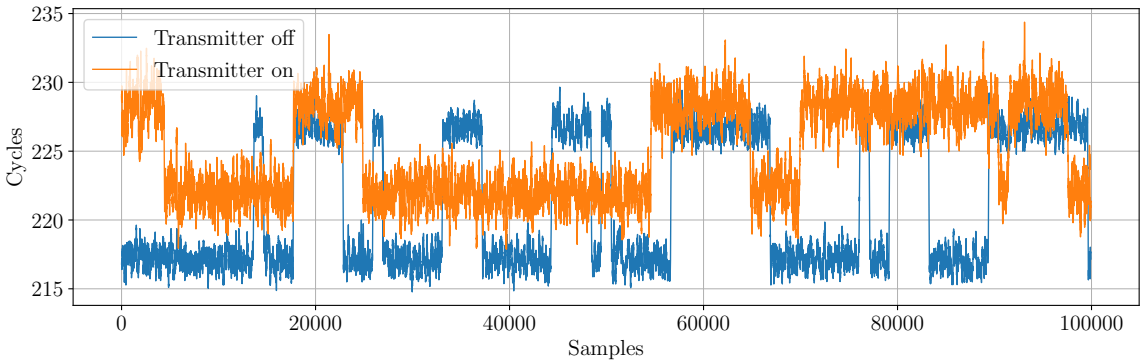

Figure 4-2: Receiver trace with the transmitter on and off.

In this section, we try to tune the following knobs in the transmitter and examine the effect it has on the receiver. For both the local EV and the remote EV, we examine the effects of 1) changing the number of L2 sets that the addresses map to, and 2) the number of addresses that map to each L2 set.

Eviction Set Size

We first examine the effect of changing the eviction set size. This can be seen in the green line in Figure 4-3 (the 1 L2 Set configuration). When changing the number of addresses mapped to each L2 set, the latency difference trend can be divided into three phases. In phase 1, when the number of addresses per L2 set is equal to or below 8, the latency difference is zero. This is because the L2 associativity is 16-way, the transmitter is unable to generate L2 misses even when using both the local EV and the target EV. In phase 2, when the number of addresses per L2 set is between 9 and 22, the latency different keeps increasing. In phase 3, when the number of addresses per L2 set is above 22, the latency difference decreases. This happens because the transmitter begins to experience LLC misses.[2] As the missing data needs to be served from DRAM, the transmitter becomes slower. The packets being injected into the network reduces in density and thus introduces less contention.

We choose an eviction set size of 20 to produce high levels of contention while reducing the risk of LLC misses.

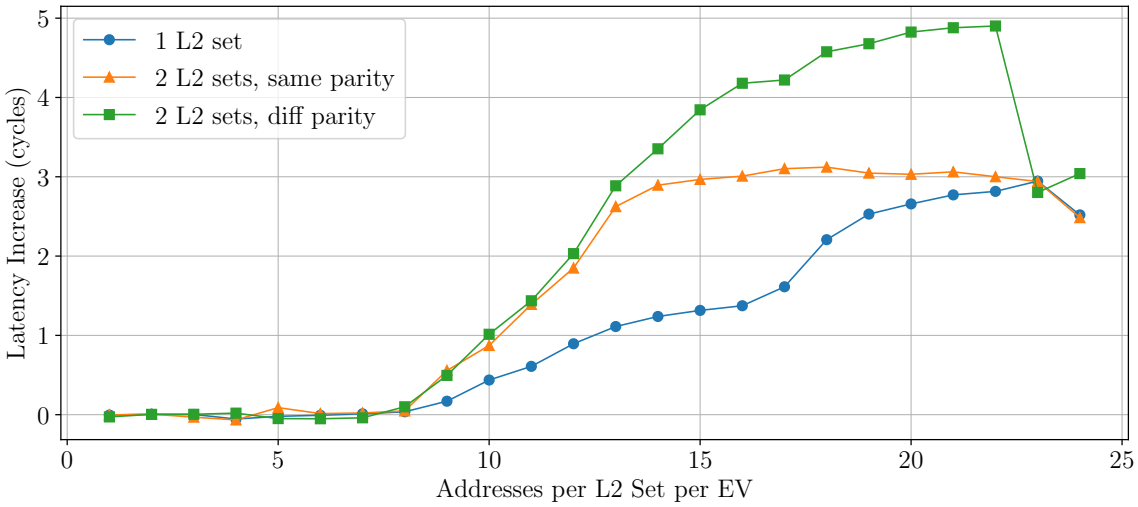

Figure 4-3: Increased receiver latency for various transmitter configurations.

[2]The reason for 22 addresses in one EV causing LLC misses is that the LLC slice associativity is 11-way, and addresses in one L2 set can be mapped to two LLC sets.

Number of Eviction Sets

One strategy to increase the traffic from the transmitter is to use multiple L2 sets. The base version of the transmitter uses two eviction sets that both map to one L2 set (and two different LLC slices). However, we can also add another pair of eviction sets that maps to a second L2 set. Since the loads in the transmitter are not serialized, accessing multiple L2 sets should allow for more packets in flight and therefore higher contention. In practice, we interleave the accesses to these two eviction sets to try to further increase parallelism. Indeed, when the transmitter uses two L2 sets compared to one L2 set, the receiver observes higher contention. This can be seen in Figure 4-3 with the yellow line sitting above the blue line. We have also tested that further increasing the number of L2 sets to 3 or above has negligible impacts. Our hypothesis is that when we use more L2 sets, we saturate the MSHRs (cache miss status holding registers) or some other restriction on the output bandwidth.

In addition, we also found that the parity of the set indices of the two L2 sets has an impact on the receiver's observation. Specifically, when the parity of the two sets is the same, the latency difference saturates when the number of addresses per L2 set reaches 13. However, when the parity of the two sets is different, the average latency difference keeps increasing and can reach as high as 4.5 cycles when the number of addresses per L2 set is 20. Our hypothesis is that this improved latency increase is due to the L2 internal configuration such as parity-based banking or way prediction. This form of parallelism would allow the loads from both L2 sets to be processed in parallel which leads to a higher density of packets.

4.4 Conditions for Contention

In this section, we examine the conditions necessary to cause contention and evaluate our model of the on-chip network with the observed contention results.

Recall from Table 3.4 that the eviction set pattern we use to access the LLC produces a request flow from Core $\rightarrow$ Slice, a data flow from Slice $\rightarrow$ Core, and a writeback flow from Core $\rightarrow$ Slice. This occurs for both the transmitter and the

receiver. In our contention analysis, we disregard the receiver writeback flow since the writeback occurs outside our timing window and thus will not be visible as contention.

After understanding the flows associated with LLC accesses, it would be intuitive to assume that we can cause contention between our transmitter and receiver by simply overlapping traffic flows going in the same direction on the same ring. However, we observed that this is not always the case.

4.4.1 Contention Experiments

To examine contention patterns systematically, we introduce both the transmitter and receiver and pin their cores and slices in various configurations. Figure 4-4 shows a case study performed on the top row of the processor where we fix the receiver to use Core 13 ↔ Slice 9. We then vary the transmitter by trying all combinations of cores and slices. Each row in Figure 4-4 corresponds to a core location for the transmitter and each column corresponds to an LLC slice location.

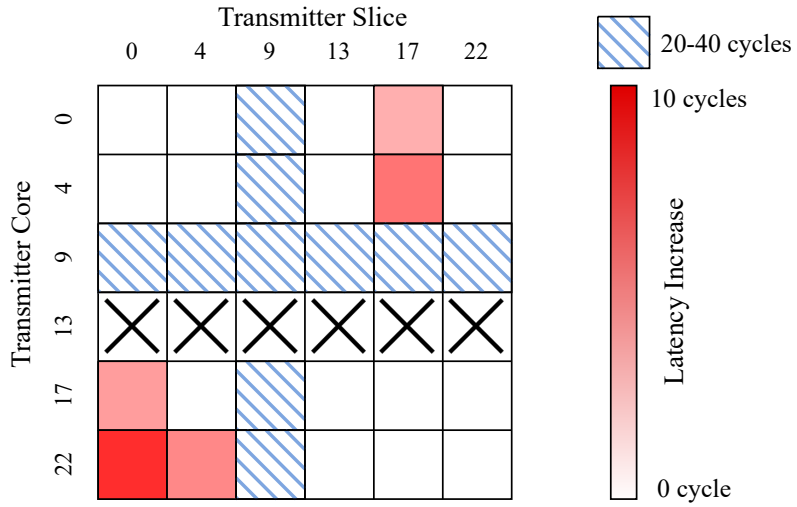

Figure 4-4: Observed increased access latency when receiver monitors Core 13 ↔ Slice 9 for various transmitter configurations.

Figure 4-4 provides a wealth of information regarding the types of contention that are observed and the conditions required for the transmitter to impact the receiver. There are a few key observations from this case study to take away.

Co-located Cores The row for Core 13 is not tested since we do not pin the transmitter and receiver to the same core. Co-locating the transmitter and receiver cores causes vastly more contention on various pipeline and private cache structures which masks any signal caused by contention on the interconnect.

Port Contention The column for Slice 9 indicates the situation in which the sender and receiver are both targeting the same LLC slice. In these cases, the receiver experiences a 20-40 cycle latency increase when the sender runs simultaneously. In the design of our transmitter and receiver, we configured the two processes to access different LLC sets to ensure that we are not observing cache line conflicts. As a result, we can conclude that the high contention levels are caused by LLC slice or directory port contention. We also observe high contention in the row for Core 9. This is also due to port contention between the transmitter's accesses to its local EV and the receiver's accesses to its monitoring set.

Flow Requirements We now focus on the other contention cases. In order for two flows to contend, they must be traveling on the same ring. That is to say that BL ring flows can contend with other BL ring flows but not with AD ring flows. The flows must also be traveling in the same direction. Due to the half-ring design of each dimension of the mesh interconnect, flows traveling in different directions will use different rings. The observed traffic flows from Section 3.3 and summarized in Table 3.4 suggest that there are two ways for the transmitter and receiver to contend. First, if the transmitter and receiver have Core-Slice pairs going in the same direction[3] and are overlapping, then they can *potentially* contend on the outgoing request which occurs on the AD ring and on the data packets returning from the LLC on the BL ring. Second, if the transmitter and receiver are going in different directions, then the LLC to core flow (i.e. the flow returning data from the cache to the core on the BL ring) of the receiver can *potentially* contend with the writeback flow (i.e. the flow of

[3]Here, we assign a directionality to a configuration by thinking of the transmitter and receiver as going from the Core to the Slice, even though traffic in fact flows both ways. For example, we say that Core 9 $\leftrightarrow$ Slice 13 goes in the same direction as Core 13 $\leftrightarrow$ Slice 17 but a different direction from Core 9 $\leftrightarrow$ Slice 4

data being written back from the core to the cache on the BL ring) of the transmitter. Note that these conditions are necessary but not sufficient to observe contention as we explain the next several paragraphs.

Priority Requirement It is not sufficient for the transmitter flows and the receiver flows to intersect. For example, consider the transmitter configuration of Core 9 $\leftrightarrow$ Slice 17 in Figure 4-4. In this configuration, the writeback flow of the transmitter travels from Core 9 to Slice 17. This should contend with the data flow of the receiver from Slice 9 to Core 13. However, no contention is observed. This is because the transmitter does not have priority. Recall from Section 2.2.3 that according to the Intel patent's description of the CMS, packets in flight get priority over packets buffered in the CMS [23]. This suggests that in order to create contention via the interconnect, the transmitter packets must be in flight by the time they reach the receiver. In this case, since the transmitter is sending packets to the right, it must be located to the left of the receiver slice. Indeed, we observe that the transmitter with configuration Core 4 $\leftrightarrow$ Slice 17 causes contention on the receiver.

Lane Requirements There are only 5 transmitter configurations that cause contention with the receiver's traffic on the interconnect, resulting in 3-8 cycles difference in the receiver's measurement. This demonstrates the multi-lane organization described in Section 3.4. From Figure 4-4, it can be seen that contention *does not* always occur even if the transmitter's and receiver's traffic use the same ring in the same direction. For example, when the transmitter uses Core 17 $\leftrightarrow$ Slice 1, its traffic overlaps with the receiver's traffic on the east-to-west direction of the Address ring, and on the west-to-east direction of the Block and Acknowledge rings. However, no contention is observed. This suggests that flows must be scheduled onto the same lane to contend with each other.

Note that traffic flowing to cores and traffic flowing to slices use opposite lanes. To observe this phenomenon, once again consider the transmitter configuration Core 4 $\leftrightarrow$ Slice 22 in Figure 4-4. In this configuration, we expect the writeback flow from

the transmitter (Core 4 $\rightarrow$ 22) to contend with the data flow in the receiver (Slice 9 $\rightarrow$ Core 13) since the writeback flow is traveling in the same direction, is on the same ring, has priority, and appears to use the same lane. However, we do not observe contention in this case because the writeback flow is traveling to a core and the the receiver data is traveling to a slice, so when determining the lane used by the receiver data flow, we need to invert the lane determined by the lane scheduling policy diagram in Figure 3-3 (a).

Chapter 5

Conclusion

The rise of multiprocessing has seen the introduction of more discrete, heterogeneous components placed onto the same processor. While chip manufacturers are striving for higher degrees of parallelism and increasing core counts, the ramifications of sharing hardware in this fashion are not yet fully understood. We have recently seen all manners of side-channel vulnerabilities based on hardware structures that are shared between processes, including the interconnects used for communication between cores. Mitigations for such vulnerabilities require a thorough understanding of the structures in question.

This thesis focused on investigating the shared network-on-chip on the latest iteration of Intel Cascade Lake processors and takes significant steps towards understanding its packet-handling behavior. We make several contributions towards the knowledge of Intel's mesh interconnect and the techniques used to probe the network. As documentation about the network is sparse, we first combine information from prior work, official documentation, and related patents to create a consolidated model of how nodes on the network are implemented. We use performance counters to reverse engineer the layout of the cores and the mappings from CPUs to physical core locations. We also determine the details about the lane scheduling policy used in the mesh interconnect to assign traffic flows to specific lanes. Next, we create tools to reliably and efficiently generate and monitor traffic on a two-dimensional mesh topology with a non-inclusive cache. These strategies require an in-depth understanding

of the caching structure and the coherence protocols used. Lastly, we use these tools to deduce timing details about contention on the network. This allows us to derive precise criteria to predict when different flows on the network contend.

This thesis lays the foundation for future work into the security implications of the on-chip interconnect. Monitoring contention patterns plays a key role in the development of covert channels and side channel attacks. Furthermore, our ability to reliably create and observe contention provide convincing evidence that the modern mesh interconnect is vulnerable to side-channel attacks. As computer architectures becomes increasingly parallelized, we believe that this work is critical to our understanding of these new vulnerabilities.

Bibliography

[1] CLFLUSH — Flush Cache Line. URL: https://www.felixcloutier.com/x86/clflush.

[2] msr(4) - Linux manual page. URL: https://man7.org/linux/man-pages/man4/msr.4.html.

[3] numactl(8) - Linux man page. URL: https://linux.die.net/man/8/numactl.

[4] RDTSCP — Read Time-Stamp Counter and Processor ID. URL: https://www.felixcloutier.com/x86/rdtscp.

[5] sched_setaffinity(2) - Linux manual page. URL: https://man7.org/linux/man-pages/man2/sched_getaffinity.2.html.

[6] Intel® Xeon® Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual. page 236, July 2017. 336274-001US.

[7] Intel® 64 and IA-32 Architectures Optimization Reference Manual. page 868, May 2020. 248966-043.

[8] Onur Aciiçmez. Yet another microarchitectural attack: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, CSAW '07, page 11–18, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1314466.1314469.

[9] Ian Cutress. Intel Announces Skylake-X: Bringing 18-Core HCC Silicon to Consumers for $1999. URL: https://www.anandtech.com/show/11464/intel-announces-skylakex-bringing-18core-hcc-silicon-to-consumers-for-1999.

[10] Ian Cutress. The Intel Core i9-7980XE and Core i9-7960X CPU Review Part 1: Workstation. URL: https://www.anandtech.com/show/11839/intel-core-i9-7980xe-and-core-i9-7960x-review.

[11] Johan De Gelas Cutress, Ian. Sizing Up Servers: Intel's Skylake-SP Xeon versus AMD's EPYC 7000 - The Server CPU Battle of the Decade? URL: https://www.anandtech.com/show/11544/intel-skylake-ep-vs-amd-epyc-7000-cpu-battle-of-the-decade.

[12] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[13] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Trans. Archit. Code Optim.*, 13(1), March 2016. doi:10.1145/2870636.

[14] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. *CoRR*, abs/1507.06955, 2015. URL: http://arxiv.org/abs/1507.06955, arXiv:1507.06955.

[15] Natalie Enright Jerger, Tushar Krishna, Li-Shiuan Peh, and Margaret Martonosi. Morgan & Claypool, 2017. URL: https://ieeexplore.ieee.org/document/7987470.

[16] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987, Fukuoka, October 2018. IEEE. URL: https://ieeexplore.ieee.org/document/8574600/, doi:10.1109/MICRO.2018.00083.

[17] Paul Kocher, Jann Horn, Anders Fogh, and Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[18] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[19] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[20] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015. doi:10.1109/SP.2015.43.

[21] John D McCalpin. Mapping Core and L3 Slice Numbering to Die Location in Intel Xeon Scalable Processors. page 17.

[22] John D. McCalpin. Address hashing in Intel processors. In *IXPUG*, 2018.

[23] Andres Mejia. Anti-starvation and bounce-reduction mechanism for a two-dimensional bufferless interconnect, August 2016. URL: https://patents.google.com/patent/US9407454/en.

[24] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/paccagnella.

[25] Colin Percival. Cache missing for fun and profit. 08 2009.

[26] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 565–581, USA, 2016. USENIX Association.

[27] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptol.*, 23(1):37–71, January 2010.

[28] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 39–54. IEEE, 2019. doi:10.1109/SP.2019.00042.

[29] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 473–482, 2006. doi:10.1109/ACSAC.2006.20.

[30] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 159–173, Bellevue, WA, August 2012. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu.

[31] Wenjie Xiong, Stefan Katzenbeisser, and Jakub Szefer. Leaking information through cache lru states in commercial processors and secure caches. *IEEE Transactions on Computers*, 70(4):511–523, 2021. doi:10.1109/TC.2021.3059531.

[32] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. 2019.

[33] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179, 2018. doi:10.1109/HPCA.2018.00024.

[34] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[35] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015. URL: https://eprint.iacr.org/2015/905.

[36] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time RSA. *J. Cryptogr. Eng.*, 7(2):99–112, 2017. doi:10.1007/s13389-017-0152-y.

[37] Mark Zhao and G. Edward Suh. Fpga-based remote power side-channel attacks. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 229–244, May 2018. doi:10.1109/SP.2018.00049.