CLPush: Proactive Cache Transfers in NUMA Applications

by

Gautam Pathak

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Modern Non-Uniform Memory Access (NUMA) systems support a thread count of as much as 128 threads to support high performance applications. These systems usually employ a scalable cache-coherent directory mechanism to ensure that the most up-to-date data is passed around among all the cores. It is common to use invalidate-based protocols in such systems. NUMA applications incur a lot of overhead due to data not being present in a particular socket's cache and having to fetch it from a cache in another socket. For example, in applications such as the producer-consumer problem, when threads reside in two different sockets, having to consume data from a socket different than where data is produced can be extremely expensive. This cost occurs due to coherence messages having to cross the sockets when the consumer threads require the shared data. In this thesis, I present a cache manipulation instruction, coined *CLPush*, which proactively transfers data across to a predetermined destination, so as to reduce cache demand misses and improve performance.

The optimization is presented as an instruction hint to the processor that directs a cache to send data across to another predetermined destination. I present various variants of CLPush, which involve having one or more destinations to transfer the data to. I also discuss the potential use cases of this instruction in different applications, such as the producer-consumer problem, and Futures and Promises. I also analyse the performance of CLPush in two variants of the producer-consumer problem.

Acknowledgements

I would like to thank my supervisor, Prof. Trevor Brown, as well as Prof. Ali Mashtizadeh, and Ajay Singh for the invaluable support and inputs they gave me. I would also like to thank the Gem5 community for all their help and support. A special thanks to Gabriel Busnot and other anonymous contributors for their expert advice and support on Gem5. I also thank my family and friends for their constant unwavering support through thick and thin.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern commodity machines rely on scale in order to cope with ever-increasing computational workload requirements. In order to keep up with this increasing compute requirement, chip manufacturers have followed a general trend to increase the processor count on a machine to parallelise tasks and increase performance. This trend is seen both in general computational workloads as in the case of multiprocessor systems, as well as more specialised workloads such as Graphics, Machine Learning and Artificial Intelligence.

Modern multi-core systems appear in different architectures, often with multiple sockets each with its own set of cores with private L1, L2 caches, or via a cluster-on-die setup. In order to establish reliable inter-socket and intrasocket communication these systems employ interconnects that ensure data is transferred in a safe and reliable manner.

System scaling introduces non-uniformity and heterogeneity in the system, due to the entire system being designed in a modular manner, glued together with specialised dedicated hardware in its own right. This structure gives rise to interesting side effects that often have a large impact on performance. Non-uniform memory access, or NUMA, is one such side effect observed when the access time to memory varies depending on how memory is placed in the system.

NUMA aware algorithms often employ advanced data partitioning techniques to place data used by cores on the socket where it is used. However, due to high cross socket-latency, a large penalty is paid when threads must access data on another socket.

Software based techniques can help reduce the stress on memory transfer in a multi-threaded workload. Placing data needed by threads as close to them as possible can reduce unnecessary cross-socket communication. However, as I show in this thesis, this cannot be

completely avoided when there is certain shared data that is critical for progress, and has to be necessarily communicated across the machine when required.

Researchers have attempted to expose a better and more detailed picture of hardware execution to programmers with the means of hardware-software co-design. This approach has given rise to many specialised instructions in Instruction Set Architectures (ISAs) for things like Single Instruction Multiple Data (SIMD), cache manipulation, transactional memory and cryptography.

In this spirit, cache manipulation instructions have been introduced in major ISAs like x86 and ARM, which allow the programmer more control over hardware. Some instructions place data in such a way that other threads incur a lower penalty when accessing it. In this thesis, I contribute another cache maintenance instruction coined *Cache Line Push* also called *CLPush*, that allows the programmer to direct data from its cache to potentially any other cache in the system.

CLPush is intended to be a hardware hint that can be used in applications where threads are bottlenecked by local unavailability of shared data. I make the case for CLPush to be a useful instruction that can reduce cache misses when this shared data is requested, and improve overall performance.

This thesis is organised as follows: In Chapter 2, I explain why ensuring coherence is important for maintaining program correctness. I also give an overview of the current state of the art protocols, as well as some related work done in this direction. Chapter 3, gives a brief overview about some common shared memory applications, which CLPush would benefit. I also explain the semantics of the CLPush instruction in a program. In Chapter 4, I give a brief introduction to gem5, an open source simulator on which this work is implemented and tested, as well as a description of its components. This discussion also includes the design and testing process followed in the construction of CLPush. In Chapter 5, I describe the different implementations of CLPush, integrated into two common cache coherence protocols, MESI and MOESI. Finally, Chapter 6, Chapter 7 and Chapter 9 describe the experiment results, interesting CLPush behavior in programs, and future work respectively.

Overall in this thesis, I make the following contributions:

- I propose a new cache maintenance instruction CLPush that allows the programmer greater control over transfer of data to one or more locations in the system, all while maintaining cache coherence.
- The applications of CLPush in applications like Producer Consumer problem, and Futures - Promises are discussed.

- Three variants of CLPush for MESI protocols and two variants for MOESI are designed.
- I conduct experiments to evaluate the performance of CLPush in two producer-consumer micro-benchmarks.
- I attempt to explain the general program behavior of CLPush with regards to data placement.

Chapter 2

Background and Related Work

2.1 NUMA Systems

NUMA, which stands for Non-Uniform Memory Access is a side effect of system design where memory access time is dependent on whether the memory is present closer to the entity accessing it. Non-uniformity arises out of various factors, including but not limited to coherence messages having to travel across sockets, variable cache and memory access time, network congestion, and so on. DRAM memory is typically distributed in such a way that a group of processors enjoy far lower latency to access a bank closer to them than a bank that is placed further away physically. In order to cushion the effect of slow memory access, caches are used to provide faster lookup. Typically L1 and L2 level caches are kept private to each core, and L3 is shared among cores in a cluster/socket. NUMA systems may also allow remote cache access, for example a remote L2 cache may be accessible from a thread placed on a socket different from the L2.

Cache access latency within a NUMA socket is typically a lot faster than accessing a non-local cache. For example, in 3rd Gen Intel Xeon Scalable processors, accessing the L3 cache within the same socket has been shown to take approximately 22 ns whereas accessing the L3 cache across sockets takes approximately 118 ns [18].

2.2 Cache coherence protocols and some current state of the art examples

In a shared memory system, coherence is important as it ensures that writes to a particular memory location from a core can immediately be *seen* by other cores or entities properly (essentially creating the illusion that there are no caches). A cache coherent memory system can be seen as a blackbox that ensures all components connected to it will always see the most recently written value.

The two popular ways to handle cache coherence are to use bus snooping [13] or cache directories [27]. Traditional snooping systems broadcast all requests on a totally ordered interconnection network and all requests are snooped by all coherence controllers [30]. Directory protocols, on the other hand, use hardware structures called directories that are responsible for keeping track of processors holding a particular cache line. Directory protocols were originally developed to address the lack of scalability of snooping protocols.

As a result, this thesis focuses on directory-based cache-coherence protocols, because NUMA machines can include a very large number of processing units, and using snooping is infeasible.

2.2.0.1 Cache Coherence protocols

Cache coherence protocols are a means of using coherence hardware to ensure safe and coherent access of data. These protocols typically store a cache line *state* along with extra metadata in each cache line for bookkeeping the validity and read-write permissions. Processors initiate read/write *transactions*, which essentially consist of a sequence of *coherence messages* sent by various components such as L1, L2 or L3 caches and directories.

A *stable state* typically indicates that there is no transaction happening on this cache line at the moment. Broadly, a transaction is initiated when there is a read, write or eviction triggered on a cache line. The transaction is in effect a state machine, which dictates how a cache line's read/write permissions change until it eventually returns to a stable state.

To explain this terminology, the simplest coherence protocol, **MI**, is used, which keeps each cache line in only one of the two stable states **M (Modified)** or **I (Invalid)**. A cache line in the modified state can be read or written to (by any processor with access to the cache in question). A cache line in the invalid state is, as the name suggests, invalid, and cannot be read or written to. Consider a two-processor system, and just a single cache line

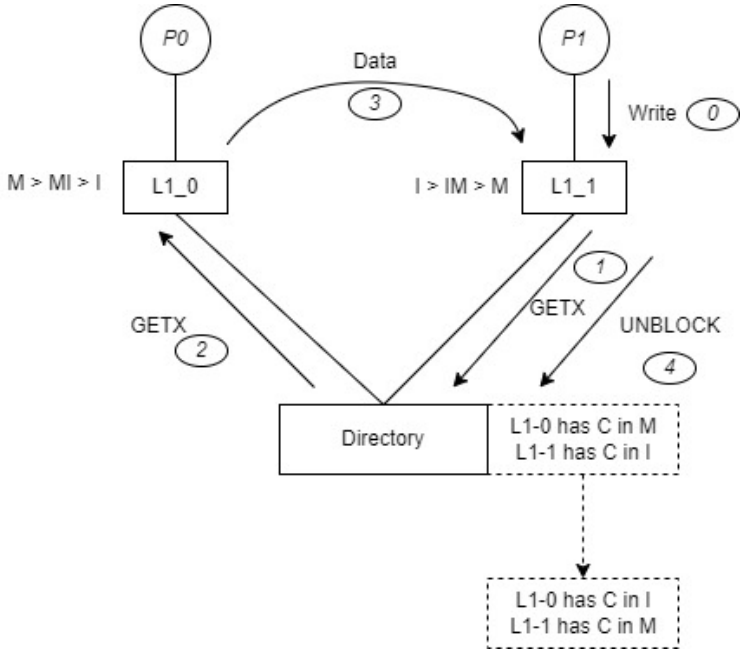

Figure 2.1: *A write transaction in MI protocol for a 2-core system with directory coherence. Messages are labelled in the order they appear.*

C (see Figure 2.1) with processor P0's L1 cache having C in the M state. The MI protocol maintains a simple invariant that if a cache line is present in the M state in any cache, all other caches must have it in the I state. Hence, only one processor can read/write to the cache line at a time.

Now, if processor P1 wants to write to the cache line, it initiates a write/store transaction at L1_1. This transaction triggers a GETX (Get in Exclusive Mode) coherence message, changing state from I to IM. IM is a transient state indicating this cache line will eventually go from I to M state. No other transaction is allowed on this cache line until this write transaction completes. This protocol helps avoid complicated race condition scenarios, and lets transactions complete one at a time. Note that computer architects are free to add optimisations to said state machine, which can help speed up the coherence protocol. The GETX message is then forwarded to the Directory that keeps track of which caches contain the cache line at any given time. The GETX request is then forwarded to L1_0, which changes state from M to I, indicating that it has lost access to the cache line, and L1_0 then sends its data to L1_1, which now updates its own state to M. The transaction is not complete until L1_1 sends an unblock message to the directory, which

finally updates its own list of caches to indicate that L1_1 now holds the cache line.

The protocols MSI, MESI and MOESI can be viewed as extensions of the simple MI protocol, where the new stable states E, S and O have the following meanings:

- S state indicates that this is a **read-only, shared** copy of the line.
- E state indicates that this cache line is **exclusive**, i.e,. is the only one present in the entire system, and has *not yet* been written to (as opposed to the M state which indicates the line *has* been written to).
- O state indicates that this cache line is one of several valid copies of the cache line in the system, but the cache in question has exclusive permission to write to it. This state allows sharing of dirty copies of data (which allows the protocol to avoid writing back to main memory in some cases).

The addition of new stable states greatly increases the size and complexity of the coherence protocol transactions (and of the overarching state machine describing the protocol). Following this introduction of cache coherence protocols, I now briefly mention a few new coherence protocols that go over and above the previous protocols to introduce optimisations.

2.2.0.2 Some newer cache coherence protocols

While different vendors use their own (often proprietary) approaches to speed up cache coherence, the scope of this thesis is limited to studying integration of CLPush in a standard MESI/MOESI protocol (where MOESI is used to evaluate the impact of "NUMAness").

Given below are details of a few coherence approaches employed in modern NUMA systems:

- Intel uses the MESIF protocol [17] for its cache coherent NUMA architectures. The MESIF protocol in short is a MESI protocol with an extra F state introduced to indicate that the cache holding a cache line in F state responds to nodes requesting data. This additional state is a little different from traditional MOESI protocols where the O state has a similar function, but there, the data can be dirty. On the other hand, the cache line in F state necessarily has to be clean. The other difference is that the F state may be silently dropped by the entity holding it, whereas the O state entity has to write the data back to memory to evict its cache line.

- ARM uses its latest CHI (Coherent Hub Interface) protocol. In short, CHI moves a little away from traditional directory/snoop interface notations and takes an approach more similar to distributed systems where a requester node initiates transactions and coherence requests are routed to home nodes that serialises transactions and establishes coherence. The lines have states corresponding to the nature of data it contains, i.e. shared, unique vs. clean, dirty.

2.3 Related Work

The interplay of memory access patterns and the underlying computer hardware plays an important role in determining the performance of the application. In a shared memory system, read and write to a particular shared location involves transfer of the cache line to and from the writer and reader in a coherent manner. If there are multiple readers involved, a writer has to invalidate all other copies, which the readers hold, before actually being able to write to it. The write, moreover, is done in the cache closest to the processor, i.e. the L1. In this scenario, periodic reads between frequent writes frequently find the cache line missing or invalidated. The readers will have to fetch the data to their own local cache, hence incurring a cache miss on every read (unless a prefetching mechanism is in use).

I introduce a custom instruction, named CLPush that leverages the programmer's mental data transfer model to proactively copy data across to the cache close to a particular core, which they anticipate to be read. Before moving to a full explanation of CLPush in Chapter 3, a number of other cache manipulation instructions are presented, which allow the programmer more control over the cache hardware.

2.3.1 Cache line manipulation instructions

Trends have shown a case for exposing caches to the programmer through the use of *hint* instructions. Three such examples are `clwb` [16], `cldemote` [14] and `clflush`, [15].

Of these three, `clwb` and an optimised version of `clflush` called `clflushopt` have found use in Intel's Optane Persistent memory. This usage case is an important example of specialised instructions being used in hardware that requires newer programming models.

`clwb` writes the cache line data in any level of hierarchy back to memory, and is like forcing memory pages back to disk in a virtual-memory system.. It may choose to retain

this data in any level of cache (for performance gain by reducing cache miss in subsequent accesses). The retention decision is done by hardware, and it may choose to invalidate the line in the hierarchy.

`clflush`, writes back the line data into memory (if it is dirty), and invalidates all copies of the line along the hierarchy. This operation frees up space in the cache for other data to reside.

`cldemote` is a hint instruction to hardware that a specific cache line should be moved from a cache closest to the processor to a level more distant from the processor. This operation also functions as a hint and may be ignored by hardware.

2.3.2 Write update protocols

Cache coherence protocols are classified into two categories, write-invalidate and write-update depending on the action taken by the protocol on writes to a cache line. Write-invalidate protocols invalidate or discard copies of cache lines held by all other caches except the cache being written to. Write-update protocols, on the other hand broadcast writes to all other other caches in a system as and when data is written to a cache line. Unfortunately, for many access patterns, update protocols generate excessive amount of update messages, which consume precious interconnect bandwidth and memory controller occupancy leading to serious performance problems [19].

There has been work done in order to integrate write-update mechanisms in cache-coherent NUMA (ccNUMA) machines. For example, Carter and Cheng [12] implement both implicit write-back from the producer side to the home node, as well as speculative updates from the home node to places that are likely to use this data. Speculative updates are done by maintaining extra hardware that detects sharing patterns, like LRU algorithms in virtual memory. Both of these mechanisms reduce remote cache misses on the reader side.

2.3.3 CHI Stashing and Push-sharing

Cache Stashing was introduced in ARM's Coherent Hub Interface Protocol (CHI) - version B. Although the terminology in this protocol is different than in standard coherence protocols, in essence, cache stashing allows a node (i.e., an individual cache or directory) in a system to put cache lines in particular nodes where it is likely to be used in the future

[3]. Of all the related works, I would say that CHI stashing is closest to what CLPush is trying to achieve.

CHI is a general coherence specification allowing development of scalable coherent systems, which can potentially combine different coherence implementations at different levels of the system. Implementations of stashing can vary widely across implementations of CHI. From the Gem5 documentation for its own CHI based implementation, CHI supports combining multiple instances of MESI and MOESI protocols and seems to support snoop requests. However, publicly available information about stashing, its uses, and its implementations, appears to be quite limited.

Modern coherence protocols including CHI are hybrids of directory based and snooping based protocols, permitting snooping at certain nodes (such as last level caches), but not at others. Stashing appears to rely on snooping in CHI, as it is permitted only at nodes where snooping can occur [2]. Additionally, it appears that cache stashing is not exposed via any instruction(s), but is instead a tool for the hardware to perform internal optimizations. I am not aware of any equivalent for x86/64.

In contrast, I give concrete implementations of CLPush for directory based MESI and MOESI protocols. In principle, implementations of CLPush could push cache lines to any node (i.e., any level of the cache hierarchy), and it does not rely on snooping.

Push-sharing or *PushSh* is another instruction hint that appears in a patent [20]. PushSh transfers data to a target node (which can be anywhere in the cache hierarchy). It also describes the state transitions a cache line undergoes (for both source and target nodes) in a typical MOESI protocol. While this patent explains various scenarios where such an instruction would be useful, it does not give a clear picture of the implementation details and nuances in race conditions that would be involved when redesigning a coherence protocol to implement such an instruction.

Chapter 3

CLPush - Proactive Data Placement

In this chapter, I introduce a cache hint instruction called CLPush. CLPush proactively places data to a particular cache on invocation of this instruction in a cache coherent manner. Since this is a hint instruction, its effects on the cache state depend on a number of conditions, which are explained in detail in Chapter 5. In this chapter, I also explore the potential benefits CLPush might offer in some shared-memory applications.

3.1 Motivation and Applications

Producer-Consumer Problem Consider a typical producer-consumer problem, where multiple producers write to a shared memory data structure, and a single consumer tries to read from it. Thinking about this from a hardware perspective, producers have to get exclusive access to a cache line in order to write to it. The consumer also has to load this line into its L1 cache in order to read from it. There are many ways to solve producer-consumer problems, using different synchronization techniques, but, fundamentally, cache lines must be transferred from producers to consumers.

Regardless of how a solution to the producer-consumer problem is implemented, significant overhead is incurred by consumers that have to fetch cache lines from lower cache levels (or from main memory), and by producers that have to send cache invalidations.

If there are a lot of producers, the consumer can become a bottleneck due to slow read progress. This problem is prominent in bounded producer-consumer problems where producers cannot write newer values as the buffer is full and old values have to be consumed first. In such cases, progress can be blocked by a slow consumer and producers are idle

until data is consumed. It would be advantageous if, while a consumer is consuming one cache line, the data to be consumed subsequently (in other cache lines) were pushed into the consumer's cache(s).

This could speed up the consumer, as it would now find the necessary data in its own cache(s) instead of incurring a demand miss in its cache(s).

As shown in the experiment analysis section in Chapter 6, producer-consumer applications stand to benefit from CLPush.

Futures and Promises *Futures* are another potential application in which CLPush can be of use. A future is a placeholder for the result of a function that is to be executed asynchronously (now or in the future). The thread that receives a *promise* executes the function and sets the return value of the function allowing the thread with the future to block (if desired) until the function is executed. The thread with the future can then proceed to other work (which may involve creating additional futures) in the meantime.

CLPush can potentially accelerate applications that use futures and promises as follows. A thread that creates a future can CLPush its arguments to a cache node where the function will be executed. A thread that executes the function can then CLPush the promise (i.e., the return value) back to the cache node where it is used. To understand why this can speed up execution, consider the following simple way of implementing futures and promises. Suppose, for example, in thread T_1 futures are accumulated in a buffer whose respective functions are to be executed by a another thread T_2. If there is some delay (due to T_2 being busy with other computation) between when a future is stored in this buffer by T_1 (and CLPushed to the thread T_2 where the computation is done), and when T_2 loads the futures from this buffer to calculate the results and set the promise, then this load is a cache hit in T_2, rather than a cache miss.

Similarly, suppose when the computation is completed, the return value is CLPushed back to an appropriate cache node, local to thread T_1 that created the future. If there is a sufficiently long delay between when T_2 computed the results, and when T_1 accesses the return value, then it is a hit in its cache, rather than incurring a cache miss.

Key Insight The key insight is that if there is a thread which is a bottleneck to the progress of a program, it forces other threads to "waste" cycles, waiting for that thread to unblock. The other threads can instead use these wasted cycles to CLPush data which is eventually accessed by the bottleneck thread. This approach turns a majority of cache misses on the bottleneck thread's side into cache hits. The cost that a thread incurs while

CLPushing data can ideally be absorbed completely by cycles that would otherwise be wasted as the thread waits for the bottleneck thread.

3.1.1 Where would CLPush be less effective?

Proactive data placement is not always possible. There has been a lot of innovation in hardware to reduce cache misses over years, including the introduction of hardware prefetchers, sophisticated cache placement policies [22], and victim caches [21] amongst others. There is no silver bullet (as of now) which eliminates network traversal and cache miss costs. So, it is also useful to understand scenarios where CLPush is less effective or even detrimental.

In the producer-consumer problem, if the consumer is relatively fast and is not a bottleneck, producers might not need to CLPush data to the consumer. If the producer's penalty for pushing of the data over to the consumer is too high, it can erode some of the gains obtained by the consumer due to cache misses. But as seen in the later chapters, the cost to CLPush data is not the same as the consumer-side cache miss cost in a simple in-order processor. Out of order processors on the other hand could see more benefits as the latency incurred by executing a CLPush instruction can be hidden by reordering CLPush with other instructions.

In general, from a hardware perspective, if the interconnect has limited bandwidth, CLPush can introduce extra congestion in the chip network due to additional coherence messages. CLPush is also ineffective where the programmer does not have prior knowledge of which cache to CLPush the data to. This scenario is possible for example, in a shared hash table, where threads insert to and read from the shared hash table. Then it is not clear as to which cache the data entries in the hash table should be pushed to.

My hypothesis is that entities which use CLPush to obtain performance gain should have the following characteristics:

- A write-then-forget scenario, where threads have to write a certain value which is then read eventually by some other thread (in NUMA, they may be on a different socket). If the data to be CLPushed is written again and again without sufficient delay, the work done by CLPush is nullified. The frequency of subsequent writes to the same cache line should allow the effects of CLPush to take place and reduce cache misses where the data is to be read.

- The programmer is sure that the data has to be **transferred eventually** across socket to be read, and that this data is necessary to make progress.

3.2 CLPush Semantics

As mentioned in the introduction, CLPush is an attempt to make data movement in the cache more accessible to the multicore programmer.

CLPush is presented as a hint instruction to the L1 cache of a particular core. It is often observed that in highly parallel use cases, the programmer knows beforehand, which logical threads use the same shared data. This shared data is often protected by locking primitives to maintain data integrity.

The format of the CLPush instruction is as follows:

```
clpush(data);
```

If the programmer foresees that this data is read by other cores, then the CLPush instruction can be used to share the data line with the other readers. As an example, this is done in the following manner:

```
int* data = new int(0);
...  //prepare the data
data = 42;
clpush(data);
```

This pushes the variable `data` from the L1 associated with the core invoking CLPush to one or more destinations depending on the implementation of CLPush. As is described in detail in Chapter 5, I have implemented multiple versions of CLPush. In summary, there is a version which broadcasts the data to all other L1 caches (even if the L1 is not referencing the data), one which copies the data to its last level cache, and another which caters specifically to NUMA systems.

CLPush has no effect on the value stored in any cache line. It may only have an effect on whether an access to that cache line is a miss or hit in a given cache node. CLPushing a cache line to a cache node nominally (in the absence of interference by other cores) causes a subsequent load of that cache line at that cache node to hit in the cache. However, interference by other cores, in the form of loads and stores to the same cache line, can cause CLPush to have no effect.

3.2.1 Effect on architectural state

CLPush is not meant to change the *result* of a program, only increase data availability in caches. Like prefetching instructions, inserting CLPush instructions in a program does not change the program semantics in any way. It only affects performance.

All of the implementations of CLPush are integrated with cache coherence protocols that guarantee total store order (TSO), and the total order on stores is established at the level of the cache directory, where stores and CLPushes on a given cache line are serialized. To make it simpler to reason about the addition of CLPush instructions to the protocol(s), coherence transactions can be initiated for CLPush instructions only when the core executing the CLPush instruction has the desired cache line in the modified (M) state, meaning it is the only core with a valid cache line that could initiate such transactions[1]. If, during a coherence transaction, some other core accesses the same cache line, this is called a coherence race. Coherence races with the transaction initiated by CLPush are systematically enumerated and addressed in the coherence protocol modifications described below. Intuitively, coherence races result in a CLPush being aborted. If no coherence race occurs, and the destination cache node has space to accommodate the pushed line, the cache line is replicated (copied) to the destination in the shared (S) state.

[1] In invalidation based protocols, if a cache line is in the modified state at any cache node then, at every other cache node, this line is either present and *invalid*, or not present at all.

Chapter 4

Gem5 and CLPush integration

4.1 System emulation using Gem5

To implement CLPush, I used Gem5, a popular open-source cycle accurate computer architecture simulator [11]. Gem5 is a modular simulator used for computer architecture research, and it encompasses both system-level architecture as well as processor micro-architecture [11].

Gem5 supports two cache coherence models. The first is called *classic memory system coherence*. According to Gem5 documentation, this model is geared towards researchers who want to study aspects of a system other than cache coherence, and require a less accurate but correctly functioning coherence model.

The second, more detailed cache coherence model is called *Ruby*. Ruby is a detailed model for studying coherence protocols, on-chip interconnects and the memory system. Ruby consists of three main components, the interconnection model, a domain specific language to model cache coherence protocols called *Specification Language for Implementing Cache Coherence* (SLICC) and the memory components, such as caches and directories.

I decided to use the Ruby model, as it allows fine-tuning the state machine for various cache coherence protocols, and gives a better picture into how CLPush can be implemented as a protocol transaction.

To model any hardware, Gem5 uses Python configuration scripts and a collection of C++ classes. Each C++ class models different parts of the system, such as caches, memory and processors. New custom components can be created by modifying these classes. These

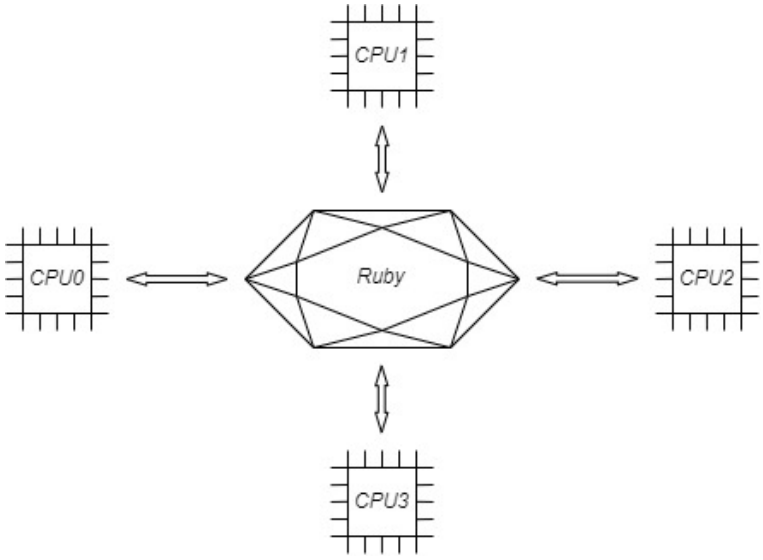

Figure 4.1: *An example Gem5 system configuration. Ruby encapsulates the interconnection network, coherence protocol and caches.*

classes, along with configuration scripts can be used to model a wide variety of custom-purpose hardware (for example, as seen in Figure 4.1 - a 4 processor system with Ruby).

I chose to modify the MESI protocol to learn SLICC, and understand the different race conditions introduced by competing coherence protocol transactions. This was followed by integrating CLPush into the much more complex MOESI protocol. Each coherence protocol Gem5 supports has its own advantages and drawbacks (Section 4.3 and Section 4.4).

4.2 Integration of CLPush into the processor pipeline

The x86 ISA was used to implement and test CLPush. It is known that certain combinations of instruction prefixes and opcodes are not supported in current x86 systems, and as such executing them on real world hardware may give undefined results. Using such unsupported combinations and implementing new instructions that use this (prefix, opcode) pair is safe in a simulator, as it is guaranteed that it does not override any existing CPU instructions. CLPush in particular uses a unique combination of Intel's special instruction prefix `repne` and opcode `mov` as the semantics of this combination is undefined in the instruction decoder in Gem5. From the perspective of an optimising compiler, CLPush will

be treated as a store, as `repne` is modifying a `mov` instruction. This is complementary to the fact that the purpose of CLPush is to send a value to a destination (cache).

X86 instructions are implemented as macro-code which are then broken down into simpler micro-ops in the processor. Each micro-op performs a simple operation, such as an arithmetic, load-store or boolean operation.

Intuitively, CLPush is most similar to a write having special side effects on caches. Hence, I modelled the micro-op `stx` (used below) upon the predefined store (`st`) micro-op. The `stx` micro-op is responsible for hinting to the cache to trigger a cache transaction for the CLPush instruction. More precisely, the instruction pipeline in a processor internally converts a `stx` micro-op into a `STX` packet and sends it eventually to a cache to trigger the transaction. **STX/StoreX can be thought of as the cache transaction triggered by the CLPush instruction and will be used often to depict CLPush in action in the transition diagrams later.**

The CLPush macro-op is defined in the Gem5 instruction decoder as follows:

```
def macroop CLPUSH_M_R {
    stx reg, seg, sib, disp
};
```

Although the `M` and `R` in `CLPUSH_M_R` indicates that this instruction involves the usage of a memory value as well as a register, in practice, CLPush makes no use of the register value, and it can be safely ignored. The `seg, sib, disp` are components of an X86 instruction and together indicate a particular location in memory (see [9] for more details).

4.3 MESI in Gem5

Following from the MI example in Section 2.2.0.1, MESI introduces two new states Shared (S) and Exclusive (E), whose characteristics were already discussed in that section. Even though popular coherence protocols like MESI or MOESI have a standard "textbook" definition [4], [5], their implementations in Gem5 have their own nuances and features, some of which are mentioned in the following sections.[1]

[1] I had initially decided to use a simpler protocol, MSI for integration on CLPush. This effort was eventually abandoned as the MSI implementation in Gem5 contained a number of bugs (and other limitations), and is regarded by Gem5 developers (as I discovered in private communication with them) as a simple teaching example that is not typically used for serious work. I thus switched to the MESI protocol, as it contained fewer implementation errors.

The Gem5 Ruby implementation of MESI has the following characteristics:

- It is an inclusive protocol, which means the L2 cache always contains a super-set of L1 cache data.
- It supports *silent eviction* of clean (which have not yet been written to, and match its copy in memory) L1 cache blocks, which means that clean read-only blocks can be silently dropped from the L1 cache without writing back to L2. This reduces the write-back traffic to L2 cache controller.

4.4 MOESI in Gem5

Gem5 offers several concrete implementations of MOESI. Out of these, MOESI_CMP_directory (CMP stands for Chip Multiprocessor) was chosen. There are other MOESI based implementations too, such as a token coherence based implementation as well as an implementation of AMD's Hammer protocol. MOESI_CMP_directory was thought to be most appropriate to implement CLPush, as it is the only implementation which is invalidate and directory based and is not tied too closely to any proprietary coherence implementations (such as AMD Hammer) with its own coherence features, or a non-directory based implementation.

An important reason for choosing this protocol over MESI was that MOESI has better support for NUMA systems. In Gem5, cores can be organised into clusters, which are essentially processor sockets. Whereas MOESI supports cache *replication* at the L2 level in clusters, MESI does not. In MOESI, L2s in different clusters can have the same data at the same time, enabling faster read look-ups at the cost of potentially higher write latency. This is because more copies have to be invalidated across clusters in order to satisfy a write to a cache line.

This facilitates an implementation of CLPush which enables a processor in one cluster to push data from its L1 to potentially any other destination cache in another cluster (socket) without having to evict data from the cluster where CLPush originates. This is important for applications where data is frequently read by cores in multiple clusters.

More specifically, MOESI supports additional coherence states in the L2 and Directory which indicate that a cache line resides in one or more clusters. Although this introduces a lot more coherence states to the state machine diagram, the coherence protocol overall has a reasonably intuitive state machine [6].

MOESI has few important characteristics related to inter-cluster requests:

- if an L1 has a cache in the exclusive state (not yet written to), and gets a write request, then the entire cache line is transferred over to the Get in Exclusive (GETX [2]) requester.
- if the cache line is in the exclusive state (not yet written to), and gets a read Get in Shared (GETS) request, the cache line is replicated over to the GETS requester.
- if the cache line has been written to, i.e. modified, and there is a GETS request, the cache line is transferred over to the GETS requester.

The action taken as mentioned in the above bullet points, that is, replication (making a copy) or transfer (moving the cache line without making a copy) depending on the state is, has a say in application performance with shared memory data structures.

For example, replication (point 2 above), can be a boon for read heavy workloads, where more entities have to read the same cache line over a period of time, and the cost of invalidation and writing is offset by the performance gain due instant read-hits caused by replication.

Whereas, what I call *hopping* (point 1 and 3 above), can be beneficial either when there is a write followed immediately by a read in the near future. Moving the entire cache line without creating unnecessary copies would save invalidation costs and speed up the subsequent write. Or when multiple reads from the another cluster are anticipated, the cache line can be transferred to the other cluster first and then replicated within the cluster itself.

4.5 Protocol modification and Testing

Ensuring the correctness of a coherence protocol is an extremely time consuming and laborious process. Not only does one have to think about all the transaction races involved, but depending on the kind of issues faced while testing the coherence protocol, design decisions may have to be made, potentially changing the performance of the coherence protocol.

Incorrect coherence protocols have been known to affect entire lineups of products. One such case of interest is a coherence bug in the Samsung CCI-400 chip bus interface

[2] **For any type of information about coherence requests, messages and actions, please refer to Table 4.1**

which appeared on Samsung Galaxy S4's system-on-chip Exynos 5410. Exynos 5410 has two sets of 4-core clusters or islands, with the first cluster consisting of 4 powerful and power-hungry cores, while the second cluster having slower, but battery saving cores [1]. Due to the bug, coherence was manually disabled on the bus and the caches were required to be flushed to main memory whenever a switch between the two CPU islands were to occur. This resulted in a significant performance and energy impact [10].

With this in mind, I tried my best to resolve any errors and used ad-hoc testing to aid in the transaction design process. This gave me some very useful insights (mentioned later in Section 5.1 and Section 5.2) and intuition in integration of CLPush in MESI and MOESI.

In Ruby, each coherence message has a certain type, such as request, response, forward and unblock. These messages travel on virtual networks. Virtual networks can be thought of as an abstraction over physical network channels and help in avoiding certain coherence deadlock conditions, mentioned in [30]. For example, a deadlock situation arises when two caches are responding to each other's requests, but the network channels are already full of other coherence requests. The caches cannot service new requests as the channels are FIFO in nature, and they will block until the responses are successfully pushed. But the response messages cannot be pushed, as the channels are already full. This example is taken from [30], which has a more in-depth discussion on deadlock avoidance as well.

Yet, there can be protocol races, due to a message from different virtual networks arriving at a cache or directory in a late or early fashion. In order to test sources of errors arising out of this, I decided to use an inbuilt random tester in Gem5 called **Ruby Random Tester** [3].

For the purpose of verifying correctness, I made modifications to the Ruby Random Tester that would allow injection of STX requests to any L1. I will briefly explain how the Ruby Random Tester works first. As shown in Figure 4.2, this protocol correctness tester uses a single oracle tester, called *RubyTester* connected to all L1 caches in the system.

The RubyTester is a special tester which contains *Action/Check* pairs. Each *Action* injects a store (ST) packet into the cache system, storing magic bytes (4 byte payload with consecutive hex numbers, eg. `0x40 0x41 0x42 0x43`). It also keeps a reference copy of the bytes it has stored, so that when a load (LD) packet is injected later on as a *Check* on the same address, it will validate the bytes it reads against its own reference copy.

The main advantages of Ruby Random Tester are the following:

[3]Incidentally, I found an old article published in IEEE Design and Test, Vol. 7, Issue 4 [31] which describes similar *Action/Check* pairs for testing cache coherence, and I speculate that this might be the long lost ancestor of *Ruby Random Tester*.

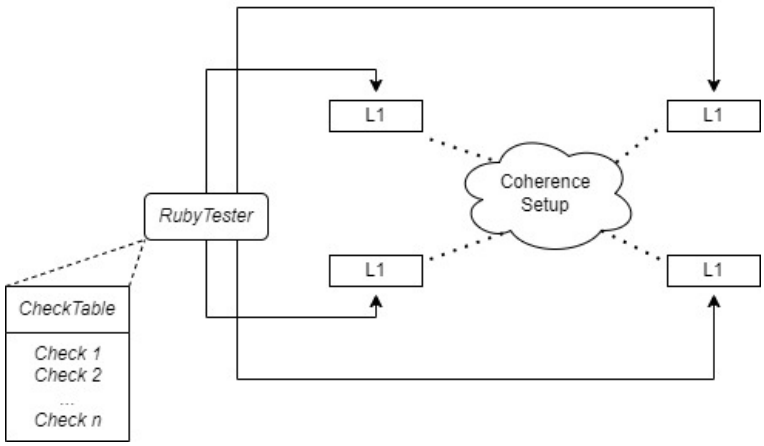

Figure 4.2: The Ruby Tester

- It generates the exact same sequence of *Action/Check* pairs in each run. This gives the ability to determine and replicate the sequence of protocol actions that led to any coherence error and reproduce coherence bugs.
- It has the option to insert random delays so that the coherence interface is sufficiently exercised.
- It has support for checking protocols in special scenarios such as false sharing. It also supports testing protocols with additional features, such as prefetching and flushing.

Now, in order to test the effect of the CLPush instruction, a STX packet is injected into the Gem5 counterpart of a load-store queue known as the Sequencer. This is done immediately after injecting a ST packet. This corresponds to the use case that a programmer will want to CLPush the cache line immediately afterwards writing to it.

The `Ruby Tester` was run many times, each time for 1 - 10 million loads and STX [4] packets, and race conditions were fixed for both MESI_Two_Level as well as MOESI_CMP_directory.

During this process, I uncovered a total of four bugs in the original protocol of MOESI_CMP_directory which were not reported before. I suspect that it is because of lack of usage of this protocol, especially in the multi-cluster setting. I observed that the focus of attention is on development of new protocols in Gem5 such as industry-supported

[4]STX is the name of packets, akin to ST and LD, but for CLPush

protocols like MOESI_AMD_hammer as well as ARM's CHI protocol. The analysis of these protocols is beyond the scope of this thesis.

4.6 Coherence errors in MOESI

Coherence transactions in MOESI_CMP_Directory proceed similarly to the description in Section 2.2.0.1. At a high level, they consist of a sequence of messages, starting with a coherence request and result in a sequence of state transitions at different caches (and possibly directories). Coherence requests (a subclass of coherence messages) are classified into two major categories. One category, GET requests, ask for some data to be present in the cache where the request originates. The second category, PUT requests, send data from the cache where the request originates to a particular location, either in memory or lower level caches. While the exact transitions for each type of request are not described in detail in this thesis, it is important to understand that all transactions have to be carefully accounted for, meaning messages and state transitions should be designed such that all transactions eventually terminate (and coherence is maintained). This involves, in part, keeping track of which response, unblock and acknowledgement messages have yet to be sent (or received) in any given transaction. Please refer to Table 4.1 for the meaning of coherence messages such as GETX, GETS, PUTX, et cetera.

The following is a description of the error encountered in MOESI_CMP_Directory, and how it was resolved. MOESI_CMP_Directory is a two level cache protocol with private L1 caches and L2 caches that are shared within each cluster (but not across clusters).

Description: Consider a four cluster system, with two cores per cluster. Each core has its own private L1 cache, and a single L2 cache in each cluster is shared between the two cores in each cluster. All L2s are connected to a Directory, which tracks the presence of each cache line in the clusters. This system is shown in Figure 4.3. More specifically, Figure 4.3 depicts a race between an L2 eviction transaction (L2_repl) that performs a PUTX request, and a store transaction (ST) that performs a GETX request on the same cache line. Initially, the cache line is exclusively held in L2_2 (which means that it is the only holder of a valid copy of the cache line). The L2_repl and ST transactions begin at the same time at L2_2 and L1_7, respectively (labelled as number 1). These two transactions race with one another. Individual messages are shown in the figure, annotated with the order in which they occur (if the number assigned to two messages is the same, they occur concurrently).

The following is a description of what happens in detail. The store misses in L1_7, and sends a GETX (labelled, number 2) message to the parent L2_3 to see if the cache line is

Coherence	
Coherence Message	Description
GETS	request for shared permissions to satisfy a CPU's load or IFetch.
GETX	request for exclusive access.
StoreX	request to CLPush data to hard-coded location
INV	invalidation request. This can be triggered by the coherence protocol itself, or by the next cache level/directory to enforce inclusion or to trigger a write-back for a DMA access so that the latest copy of data is obtained.
ACK/NACK	positive/negative acknowledgement for requests that wait for the direction of resolution before deciding on the next action. Examples are write-back requests (these messages have WB_ pre-pended to them), exclusive requests.
PUTX	request for write-back of cache block. Some protocols (e.g. MOESI_CMP_directory) may use this only for write-back requests of exclusive data.
PUTS	request for write-back of cache block in shared state.
PUTO	request for write-back of cache block in owned state.
PUTO_Sharers	request for write-back of cache block in owned state but other sharers of the block exist.
UNBLOCK	message to unblock next cache level/directory for blocking protocols.
L1_Replacement	request to evict a cache line from L1 cache. Triggers one of PUTX/PUTS/PUTO request.
L2_Replacement	request to evict a cache line from L2 cache. Triggers one of PUTX/PUTS/PUTO request.

Table 4.1: Coherence protocol messages (Taken and modified from Gem5 documentation).

present there. L2_3 does not have the cache line, so upon receipt of this GETX, L2_3 sends a GETX message (number 3) to the directory to check if other clusters/memory has the cache line or not. While it waits for a response, it blocks and stops responding to requests for that cache line that are not related to this transaction. Blocking makes it easier to reason about the protocol, since otherwise one would have to imagine more complex state transitions as multiple different transactions are interleaved. The directory knows the line is present at L2_2, and so it forwards the GETX message (number 4) to L2_2 and indicates that L2_2 should transfer its data to L2_3. Similarly to L2_3, the directory blocks incoming requests for this line until this transaction is completed. Next, L2_3 receives this cache line and transfers it to L1_7 (number 5) to complete its store request. After the store is able to

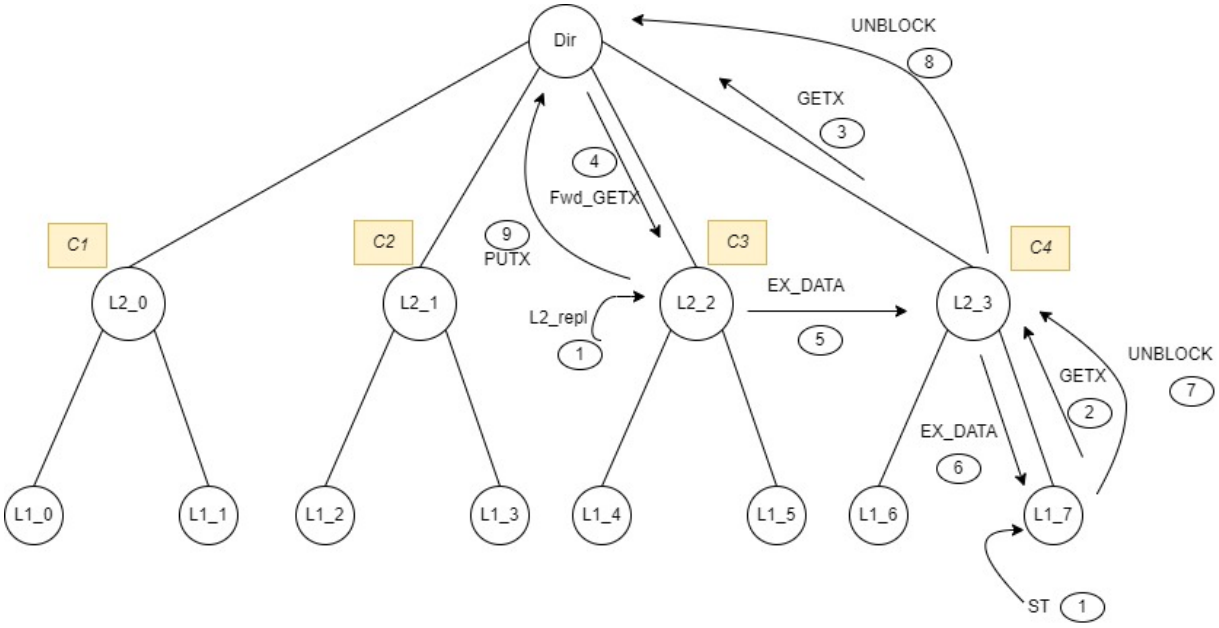

Figure 4.3: Fwd_GETX - PUTX Race

complete, L1_7 still has to unblock L2_3 (number 7) which is in a blocked state, after which L2_3 will be able to service newer transaction messages for that cache line. The L2_3 will in-turn unblock the Directory (number 8), hence fully completing the Store transaction.

The ST transaction "wins" in this case, and the exclusive data held by L2_2 is handed off to L1_7 via L2_3. But L2_2's *PUTX* transaction is still unresolved. This is a potentially dangerous situation where coherence can be broken due to an unaccounted transaction. As far as the directory is concerned, the GETX transaction has completed successfully and cluster C_3 is now in sole possession of the cache line.

Normally, when a PUTX request arrives at the Directory (number 9), it is asking the Directory for permission to write-back its data. The Dir then responds with a *WB_ACK* or *WB_NACK* (which is permission to *write-back*, not labelled in the figure). The L2 then responds back with data, or tries a *PUTX* transaction at a later point of time depending on whether it was Acked or not.

But in this particular situation, the data was already handed off to the requester. Since L2_2 has serviced the GETX transaction, it knows that it no longer holds the cache line. Hence, L2_2 simply responds back with an *Unblock* (had it still been the owner, it would have responded with the data) to the Directory to nominally complete the PUTX request

which it had previously sent out. The behavior of the Directory when such kind of PUTX-GETX race happens was not originally accounted for in the protocol. I fixed the protocol to indicate that an Unblock is an implicit indication to the Directory that some kind of ownership transfer has occurred and it should simply complete the *PUTX* transaction, as the ownership accounting has already been done beforehand in GETX.

Note: the other three bugs were similar to the previous one, but involved putting back a cache line in *Owned* state and a racing *GETS* transaction.

Chapter 5

CLPush variants

A StoreX request differs from a GETS or GETX request, as it requires the sender to *push* the data to some other location, rather than *request* data. Based on the current implementations described in this thesis below, the intended way to use a CLPush instruction is to write to a cache line and then immediately CLPush it. Correspondingly, in these implementations a CLPush instruction can only succeed if a cache line being pushed is in the modified (M) state in the L1 cache where CLPush is being performed. This is because intuitively, the modified state is the state where the L1 cache performing CLPush has the most up-to-date value.

In most invalidate based coherence protocols, this indicates there is no other copy of this cache line in any other location which can write to it. Clean copies can be kept, but they must either be invalidated, or the modified cache line must be written back to it. A different implementation of CLPush could very well be imagined, which pushes from a different state, such as the shared (S) state. However, for this case, things get simplified with the modified state because of the reduction in number of cases which have to be considered while designing the actual transaction performing the data push. I identified this as a natural use-case, but there could very well be other use-cases which are as useful (if not more) than this.

In this chapter, I explain the 5 implementations of the CLPush instruction. Of these, three implementations of CLPush are implemented in MESI, each having different destination cache(s) to push the data to. The remaining two are implemented in MOESI.

I give a correctness sketch and describe the race conditions involved in detail for Section 5.1.1 (MESI - Broadcast) and Section 5.2.2 (MOESI - Coupling). Section 5.1.3 (MESI

L2push) and Section 5.1.2 (MESI - Single L1) are simpler extensions of Section 5.1.1 (MESI - Broadcast).

5.1 MESI based implementations

I started with a simple single socket/cluster implementation for the MESI protocol. It is assumed that all cores are part of one single cluster. In all MESI versions (see Figure 5.1), each core (not shown in figure, but assumed to be connected to each L1) has a private L1 and all cores share a single inclusive, shared L2.

5.1.1 MESI - Broadcast

This implementation was designed keeping in mind that there may be more than one destination cache which require the same data. The data is broadcast to all L1 caches on the cluster except the L1 initiating CLPush.

Fast Path: In the best case scenario, i.e. the fast path, there are no conflicting transactions on the same cache line, and CLPush succeeds in pushing the data to the destination as intended.

As seen in Figure 5.1 (a), a StoreX request is sent from L1 to L2 (labelled number 1, in sub-figure (a)). L2 will then broadcast this message to every L1 except the requester (number 2). For a given L1 cache, when the data reaches this cache, it will either store the data and send an *ACK* message back, or it will reject the pushed data and send a *NACK* message (number 3). The scenarios in which the data is rejected are described in further detail below.

The requester (L1 cache) tracks future sharers of the block by maintaining a sharer list (not shown in Figure 5.4) based on who *ACKed* or *NACKed* the request. Upon completion of the transaction, an unblock message along with the sharer list is sent back by the CLPushing L1 to the L2 cache (number 4). This ensures that the L2 has an up-to-date list of sharers, as well.

The fast path represents an ideal scenario, where there are no competing transactions, or evictions. However, in reality, there might be overlapping transactions in progress, or cache capacity issues, which hinder the progress of CLPush.

Below I give a step-by-step explanation of a selection of the races that can occur, and what actions are taken to resolve these races at different points of time in the transaction.

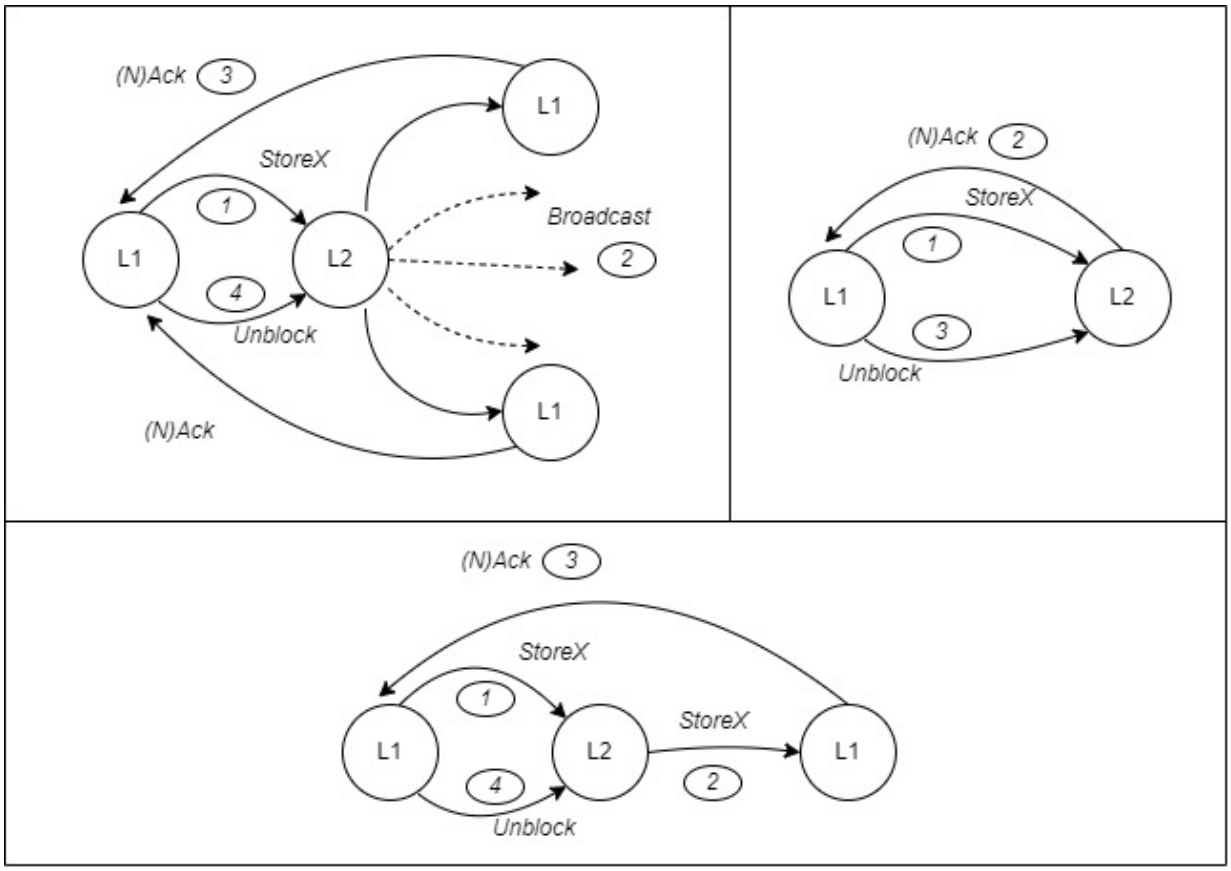

Figure 5.1: *MESI Fast Paths. MESI-Broadcast (a), MESI-L2Push (b), MESI-singleL1 (c)*

The races for other MESI variants are not discussed as they are similar to those discussed here. For diagrams, see Figure 5.2.

Let the cache line to be CLPushed be called C, the L1 trying to CLPush data $L1_p$, and the destination L1s $L1_{d_1}...L1_{d_n}$.

CLPush fails before sending a StoreX message : In the first race, suppose a core writes to a cache line C, and then before it can CLPush the cache line, another core sends a GETS or GETX request that changes the cache line's state (labelled number 1, 2 in sub-figure (a)). Since the state is no longer M, the L1 performing the CLPush no longer has the most up-to-date value of the cache line, so the CLPush simply aborts and has no effect.

A similar scenario can arise if a core writes to a cache line C, and then before CLPushing it, performs another access on a different cache line, causing C to be evicted from the L1. This does not conform to the recommended way to use CLPush, but I nevertheless wanted to clarify what can occur if any cache accesses are performed between a write and subsequent CLPush of the written cache line. (Note that the other cache line accessed would need to map to the same cache associativity set as C in order for this to occur.) Since the L2 is inclusive, the same thing can occur if C is evicted from the L2, and an invalidation message is thus sent to $L1\_p$.

In both cases, the other transaction that interfered with the CLPush proceeds as usual (i.e., the same as it would have if CLPush were not used at all).

L2 receives GETS/GETX/L2_Replacement before the StoreX message : In this case, priority will be given to the request that reaches the L2 first, specifically the GETS/GETX/L2_Replacement message (labelled number 1 in sub-figure (b)). The L2 will forward this request to the StoreX sender (number 2), as it is the only valid owner of the cache line (by virtue of M state).[1] Since StoreX has already been initiated from this L1 (number 3), it will stall the GETS(X) request until it receives a NACK from the L2 (number 4), and the GETS(X) transaction will proceed. This has the disadvantage of added stalls as StoreX presents an additional overhead for the other transaction until it receives a NACK.

Similarly, since MESI is an inclusive protocol, a replacement (due to capacity issues) at L2 will trigger an invalidate message to be sent to $L1\_p$. The case where the invalidate message is sent to $L1\_p$ *before* the StoreX is sent is described already

[1] Note that the cache line must still be in M state at $L1\_p$, since it was M when the StoreX message was sent, and $L1\_p$ blocks incoming messages until the L2 has responded with an ACK or $NACK$ message.

above. So suppose it is sent after the StoreX message is sent. In this case, since $L1\_p$ blocks incoming messages until it receives an *ACK* or *NACK*, the invalidate message will be blocked at $L1\_p$ until the L2 receives the StoreX message, and sends a NACK back, causing the CLPush to be aborted.

L2 receives GETS/GETX/L2_Replacement after StoreX message : In this case, priority is given to the CLPush transaction, and the GETS(X) or replacement request is stalled at the L2. After broadcasting the data, the StoreX will reach the L1 whose request is stalled at the L2 (number 3 in sub-figure (c)).

In this case, the $L1_c$ will simply NACK the StoreX request (number 4). There may be one less sharer, but StoreX is still broadcast to as many entities as possible.

StoreX races with a previous store at $\mathbf{L1_p}$: In this scenario I observed that a race could occur due to a previous Store transaction from $L1_p$ not being complete. As mentioned in 3.2.1, StoreX should only be initiated when the cache line is in modified state. Thus when a previous Store transaction that had succeeded in the past for $L1_p$ sends an unblock to L2 (labelled number 2 in sub-figure (d)), sending a StoreX from the same cache could introduce a race ($L1_p$ has already got the data to write, hence it is free to issue a StoreX after getting the exclusive data). This can happen as Unblock and StoreX messages are sent on different virtual networks. To fix this race, if the StoreX reaches L2 when it is in a transient state (waiting for the Store to finish), then the StoreX will stall until the previous Store transaction is completed.

5.1.2 MESI - Single L1

This transaction is similar to Section 5.1.1, with the difference that the data is pushed to a single target L1. For our implementation, I have hard-coded the destination L1 cache for simplicity. In practice, the target cache could be specified via a register or instruction argument.

5.1.3 MESI_L2Push

This transaction is an even more trimmed down version of the previous protocols. In a way, it is similar to `cldemote` [14] or `clwb` [16] as it involves writing back to a lower level cache. The L2 will ACK or NACK the data depending on whether there are any race conditions or not. The requester L1 sends an *Unblock* to end the StoreX transaction.

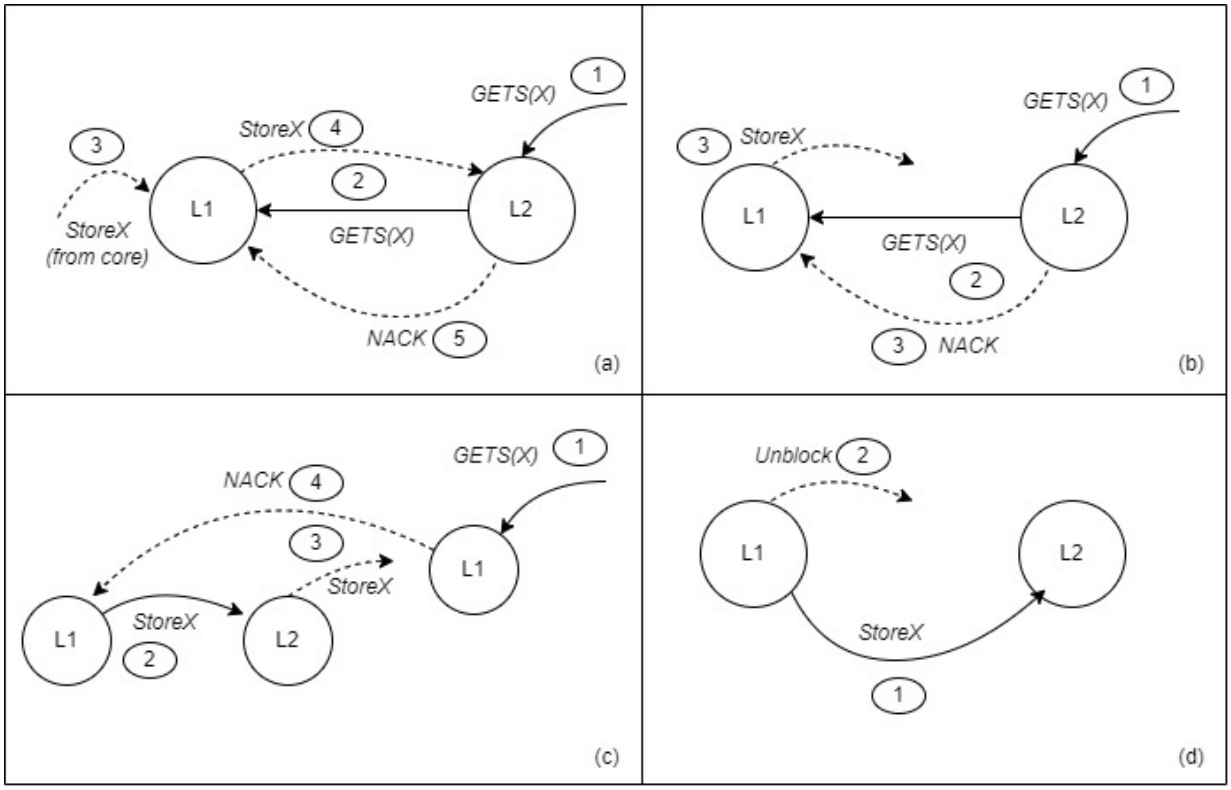

Figure 5.2: *Races in MESI_L1ack CLPush version*

5.2 MOESI based implementations

As mentioned in Section 4.3, MOESI contains many more coherence states, and an *Owned* state optimisation. It also supports non-inclusivity, and L2 replication across clusters. Since MOESI in Gem5 supports the concept of clusters, this complicates the design of the CLPush transaction significantly. I started with a preliminary naïve version, which was significantly harder to design than the coupling version.

5.2.1 Naïve MOESI

The CLPush transaction in this protocol is modelled after a typical Store transaction in MOESI, with some differences. A typical successful store transaction was mentioned in Figure 4.3 where the successful store was racing with a PUTX.

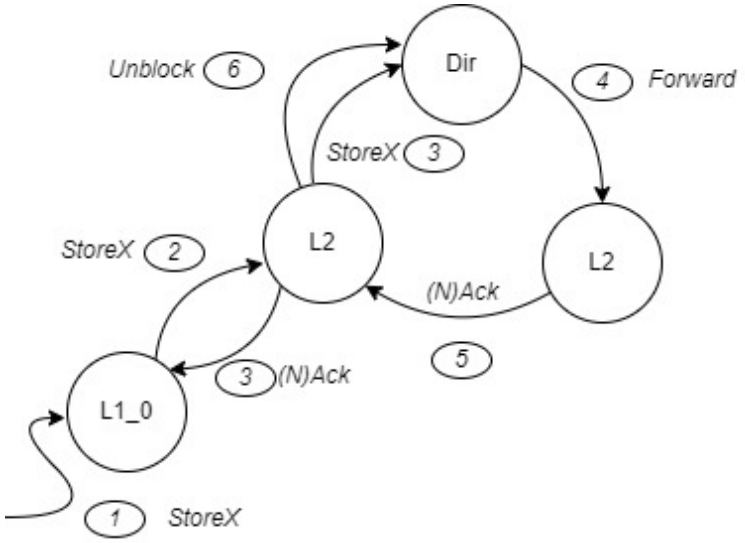

Figure 5.3: *Stages of CLPush for MOESI - naïve (order of events accd. to numbering)*

The way in which CLPush proceeds is visualised in Figure 5.3. The L1 which wants to push data sends it to its cluster's L2 (which sends back an Ack indicating that data was received, labelled number 1). This L2 then passes it on to the destination cluster's L2 via the Directory (number 3). In order to complete the transaction, the destination L2 sends a *ACK/NACK* (number 3) to the source L2 (depending on whether it accepted the data or not), and the source L2 unblocks the Directory.

The key difference between a store and CLPush transaction is that in a CLPush transaction the L1 requester starting the CLPush is immediately unblocked as soon as it gets a (N)Ack. This is opposed to the store transaction, where the L1 starting store has to wait until it gets the exclusive data.

The idea is that two successive caches can be paired in the form of request - (N)Ack is extended and used in Section 5.2.2.

The main disadvantage of this implementation is that three entities (source L2, destination L2 and Directory) are blocked while the transaction proceeds. It is also much more difficult to think about the races involved.

5.2.2 MOESI Coupling

The MOESI coupling variant follows a simple $request \longleftrightarrow (N)Ack$ handshake approach to send the cache line to a particular destination. In this implementation, a cache line will be shared with the destination in a series of request-(N)Ack request response messages. The cache/directory receiving a StoreX message may accept or reject it, depending on conditions like racing coherence transactions or capacity issues. In the case of successful receipt of a StoreX message, the cache/directory blocks until it receives a positive or negative acknowledgement from the next cache:[2].

The MOESI coupling variant has significant advantages over the previous naïve version:

- It strives to keep the caches and directory involved in a stable, non-blocking states as quickly as possible with the *ACK/NACK* approach, so that cores which want to read a cache line can access the caches effectively (both inter-cluster and intra-cluster cores). Note that in this implementation, after a successful transaction, only the CLPush requester and the destination L2 cache end up sharing the cache line. No other cache/directory act as sharers, even if the CLPush transaction fails midway (see Figure 5.4).
- In case a racing transaction approaches the path of traversal of StoreX first, the CLPush transaction will be thwarted such that there are a very few number of blocked elements.
- It is far easier to reason about races as it follows a simple *ACK/NACK* approach.

The following is a description of the CLPush transaction, with a step-wise explanation, with potential race conditions explained along the way (please refer to Figure 5.4 a, b, c, d).

Step 0: A CLPush transaction is initiated from the core. If L1_0 is idle (i.e., no transactions on that cache line), it prepares to send out a StoreX message to L2_0. However, before the StoreX message is sent out, a potential race condition can occur. Specifically, a competing coherence message (triggered by a concurrent coherence transaction) for this cache line can reach L1_0 before the StoreX message is sent out. In this case, the CLPush transaction will be aborted.

[2]For an explanation of how this implementation works, and how races are resolved, it is not necessary to have a full context of the states involved. However, if the reader is interested in knowing more about the coherence states, they may refer to [6] for a full description and meaning of cache coherence states.

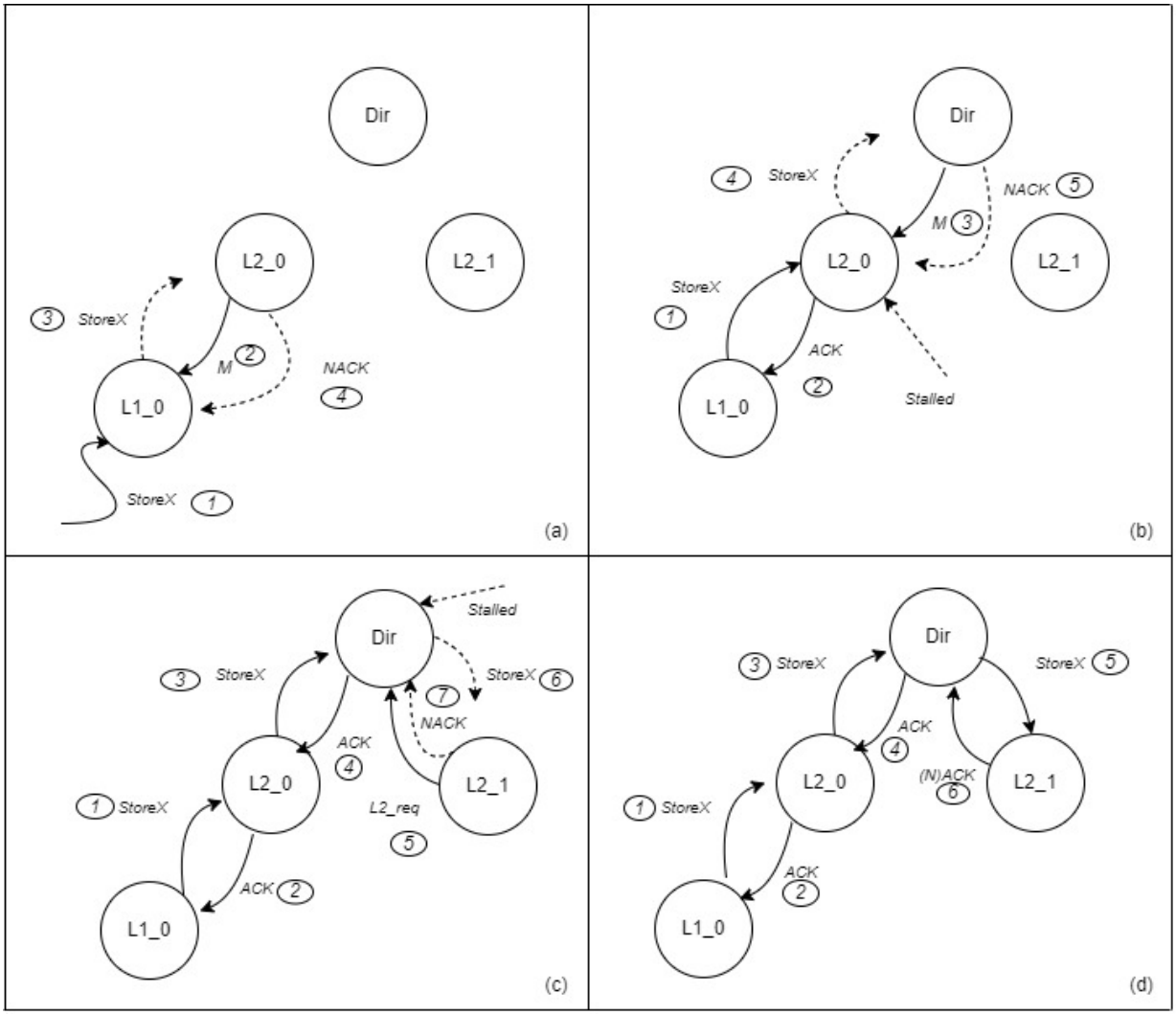

Figure 5.4: *Stages of CLPush with potential races. Step 1 (a), Step 2 (b), and Step 3 (c), Step 4 (d)*

Step 1: StoreX is successfully sent out from L1_0 (labelled number 3 in sub-figure (a)). If there is no racing transaction, L2_0 will ACK this message and the CLPush transaction will proceed from L2_0.

In case there is a another transaction T whose message M reaches L2_0 before the StoreX, there is a race condition. L1_0 has already switched to a transient blocking state, because as far as it is concerned, it has started a CLPush transaction. Since L2_0 has

forwarded M (number 2), it will be in a transient state as well. Hence, when the StoreX message reaches L2_0, it will NACK the StoreX message (number 4) as it will give precedence to T, whose message M it had received first. The transient state of L1_0 will fall back after receiving the NACK to the stable state it was in before the CLPush was started, and the racing transaction T will proceed to be completed.

Step 2: StoreX arrives at L2_0 successfully (labelled number 1 in sub-figure (b)). If there is no racing transaction, the StoreX will reach Directory and the Directory will ACK the StoreX back.

But if L2_0 gets a racing transaction T while StoreX is sent out to the directory (number 4), there is a race condition. L2_0 has already switched to a transient blocking state, because as far as it is concerned, it has started a CLPush transaction. L2_0 also sends an ACK to L1_0 (see previous point for successful receipt of StoreX by L2_0).

There are two scenarios in this stage (possibly happening together). T can come from another L1 within the cluster or from the directory, or both. In the case of it coming from L1, its request will be stalled at L2_0, allowing the CLPush transaction to proceed.

In case it comes from the directory, it implicitly means that a message from T has reached the directory first, and the CLPush transaction will not proceed further (as the Directory will NACK the StoreX message from L2_0 (number 5)).

Step 3: The StoreX message arrives successfully at the directory (labelled number 3 in sub-figure (c)). The directory will send ACK to L2 (number 4) indicating that the StoreX message has reached it. The directory will forward the StoreX message to L2_1 (number 6) and meanwhile block all incoming requests to the same cache line from other clusters.

Note that this is a possibly dangerous scenario, when transactions can start from a source cluster as they are moving back to stable states immediately after getting an ACK. However, this is safe as the caches transition into stable states that indicate there may be sharers outside the cluster. For example, in the Gem5 implementation [6], L2 stable states, such as ILS, ILOS and ILO, indicate that cache lines may preside in other chips. I reuse some of these stable states to indicate that coherence is not blocked anymore, yet ensuring coherence transactions now involve cross cluster invalidation if writes occur. This will let read transactions happen within cluster, but a store will have to necessarily pass through directory, to invalidate any copies that might possibly exist in other clusters.

Step 4: StoreX arrives at L2_1 (number 5 in sub-figure (d)). L2_1 can either reject the request by sending back a NACK (if it is full, or there is a racing coherence request for the same line on L2_1), or accept the data, write it in Shared state, and send back an ACK (number 6).

Chapter 6

Experiments

The MOESI-coupling implementation of CLPush was tested on 2 microbenchmarks in order to understand how CLPush affects performance in each of these microbenchmarks (see Section 6.2). Both microbenchmarks model the producer-consumer problem in different ways. The first microbenchmark, called *round based producer-consumer* uses a contiguous array as a shared data structure to model the producer-consumer problem. The second microbenchmark called *Doubly locked queue*, uses a concurrent queue with two locks as a shared data structure to model a dynamically allocated pointer based shared data structure for the producer consumer problem.

6.1 System Topology

For my experiments, I modelled multicore NUMA systems in Gem5's inbuilt system topology called MESHDirCorners_XY.

As seen in Figure 6.1, MeshDirCorners_XY arranges all the cores, caches, and directories into a two dimensional grid (or mesh), allowing the programmer to include as many cores in a $m \times n$ grid. The cores are allowed to be organized into as many clusters as the number of rows. The implementation organizes the cores in a cluster into a row, connected in a line. The reader might worry that this would impose unreasonable costs on communication between cores in a cluster, but I address this by modifying link latencies, as described below.

As the name suggests and as seen in Figure 6.1, in MeshDirCorners_XY, Directories (which act as an interface to the memory) are present only on four corners of the grid.

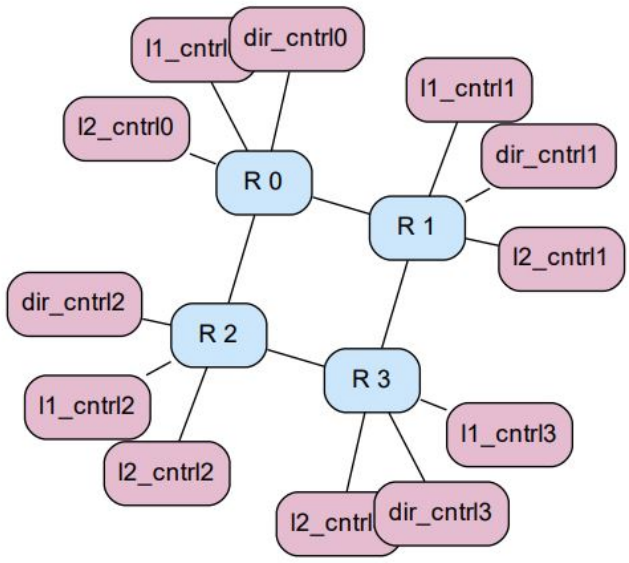

Figure 6.1: *MeshDirCorners_XY setup with 2clusters, 2threads per cluster*

Each individual cluster is present as a row (eg. (R0, R1) being a row, as well as (R2, R3)), and each L2 is *shared within*, but *private to* the cluster. Each core (not shown in the figure) has an L1 totally private to it.

$R0, R1, R2...$ are routers which route the coherence message from a source cache to a particular destination cache. In the figure, components connected to $R0$ and $R1$ form the first cluster, and the second cluster comprises of components connected to $R2$ and $R3$. For the first cluster, `l2_cntrl0, l2_cntrl1` together represent a single logical L2. Similarly, `l2_cntrl2, l2_cntrl3` is the logical L2 for the second cluster. The processors are not shown, but they are connected to each L1 cache.

There were a few issues with the cache coherence protocol parameters and topology used for modelling NUMA. The default cache latencies did not reflect the high penalty paid when a cross socket communication has to happen, especially when threads across sockets access the same data.

In NUMA machines, the intra-socket cache hit latency is an order of magnitude smaller than a cross-socket cache hit latency. For example, an article from AnandTech, which reviews Intel 3rd-Gen Xeon Scalable (Ice Lake) processors [18] shows that the L3 cache hit

within the same socket takes ≈ 22 cycles whereas a cross-socket L3 cache hit takes ≈ 120 cycles.

Since, the coherence protocol supports only a two level cache hierarchy, L2 caches here take the place of L3 caches mentioned in [18], but the idea of having to incur a significant amount of latency when doing cross socket communication remains. This was not observed in the default parameters of Gem5. As seen in Table 6.1, the cross socket L2 write latency is just ≈ 2.5 times the intra-socket L2 read latency. Hence, it was necessary to tune the $L1 \longleftrightarrow L2$ latencies (intra-cluster) and $L2 \longleftrightarrow Directory$ latencies to better mimic the latencies observed in Table 6.2. The final tuned latencies are also mentioned in Table 6.1. The reader may wonder why I didn't use the exact latencies mentioned in [18]. The reason is that the observed latencies of caches are a result of tuning several parameters in the simulator, and the cross-socket socket write latency is the *result* of tuning these simulation parameters. Hence, these latencies are the best effort result of tuning these internal system parameters.

Observed Latency		
Cache latency	Default	Tuned for NUMA
Intra-socket Write (i.e. from local L2) in Gem5	≈ 19 cycles	≈ 19 cycles
Cross-socket Write in Gem5	≈ 45 cycles	≈ 135 cycles

Table 6.1: Observed latencies of local and cross-socket cache accesses in Gem5

Observed Latency		
Cache latency	Xeon Platinum 8280	Xeon Platinum 8380
L3 cache hit (same socket)	20.2 ns	21.7 ns
L3 cache hit (remote socket)	180 ns	118ns

Table 6.2: Observed latencies of local and cross-socket cache accesses in Gem5 [18]

After fixing these issues and fine-tuning the parameters, I was able to better mimic the latencies that NUMA systems portray, i.e. L2 intra-cluster access is just tens of cycles, and L2 inter-cluster communication takes the order of hundreds of cycles.

Writer:

```
for (int rnd=0;rnd<ROUNDS;++rnd) {
for (int i=tid;i<SLOTS;i+=nprod) {
    g.slots[i].val = rnd;
    #ifdef CLPUSH
    clpush(g.slots[i].val);
    #endif
}
__sync_synchronize();

while (g.rnd != rnd+1
&& g.rnd != ROUNDS);
}
```

Reader:

```
RDTSC(start);
for (int rnd=0;rnd<ROUNDS;++rnd) {
    int sum = 0;
    for (int i=0;i<SLOTS;++i) {
        while (g.slots[i].v != rnd) {};
        sum += g.slots[i].v;
    }
    g.rnd++;
}
RDTSCP(end);
cout << (end-start) << endl;
```

Figure 6.2: *Round based producer-consumer benchmark.*

6.2 Microbenchmarks

6.2.1 Round Based Producer Consumer Microbenchmark

This benchmark, called *Round Based Producer Consumer*, is a modified version of the Producer-consumer pattern.

Description: This microbenchmark has multiple producers spread over multiple clusters, and a single consumer present on a separate cluster. All producer and consumer threads are pinned to their processors as System Emulation Mode in Gem5 does not support thread migration.

Pseudocode appears in Figure 6.2. In this microbenchmark, a shared array of slots is written to by multiple producers. The slots are shuffled among all producers in round-robin fashion, meaning that with n producers, slots 0, n, $2n$ and so on, belong to producer 0, slots 1, $n+1$, $2n+1$ and so on belong to producer 1, and so on for the remaining producers. As a result, when each producer has written one value, the first n slots of the array are populated with values. This causes the producers to quickly outpace the consumer, and fill the array faster than the consumer can consume its contents. The consumer concurrently iterates over all of the slots of the array, repeatedly reading the current slot until it successfully consumes a value before moving onto the next slot.

This is done for a fixed number of rounds. The current round number is stored in global

Simulated System	
Parameter	Value
CPU	BaseTimingSimpleCPU
No. cores per cluster	2, 4, 8, 16
No. of Clusters	4
L1D, L1I size	32kB per slice, 8-way assoc.
L2 size	1MB per slice, 16-way assoc.
Topology	MeshDirCorners_XY

Table 6.3: Parameters of the system in simulation.

variable *g.rnd*. In each round, the producers fill the array, with each producer writing the current round number, and the consumer reads all values in the array, one by one (ensuring that it sees the current round number in each slot before proceeding). Once a producer has reached the end of the array it spins until *g.rnd* is incremented. The consumer increments this *g.rnd* when it finishes reading all values for the current round, indicating that a new round has begun.

The simulation parameters are mentioned in Table 6.3.

6.2.2 Experiment Methodogy

Three experiments were conducted, based on:

- varying thread count per cluster (TPC), and keeping rounds and array size fixed
- varying the number of rounds, and keeping TPC and array size fixed
- varying the array size, and keeping TPC and rounds fixed

The first experiment was run for $TPC \in \{2, 4, 8, 16\}$ with array size 1024, for 1000 rounds with 4 clusters. The second experiment was run for $rounds \in \{10, 100, 1000, 10000\}$ with array size 1024, 4 clusters and 4 TPC. The third experiment was run for $arraysize \in \{12, 64, 256, 512, 1024\}$ for 1000 rounds with 4 TPC. Each experiment is run with two different implementations of the benchmark, one in which each producer performs CLPush after storing a produced value (CLPush), and one in which no CLPush instructions are performed (NoCLPush).

For each experiment, parameters like number of CLPush instructions, time taken by the consumer (in ns), and the percentage of CLPush instructions that are aborted (and have no effect) were measured. We record the total time required to perform all of the rounds, excluding the setup and tear-down time for the benchmarks. In each experiment, array slots were 64 bytes each to avoid false sharing. These experiments were run in Gem5's System Emulation (SE) mode, with the producers pinned on every cluster except the second cluster (cluster 1), and the single consumer was pinned on the first core of this cluster. SE automatically assigns a core to each thread according to the order of thread creation, and does not do thread migration. Along with this, parameters such as latency for loads, stores, and STX were also measured by looking at execution traces and manually calculating latency for a small number of reads, writes and CLPushes (see Table 6.4).

Observed Latency (in cycles)		
LD/ST/STX action	CLPush	NoCLPush
ST by producer	130-140	145-155
LD from consumer	26-30	137-150
STX response time	14	-

Table 6.4: Observed latencies of Loads, Stores and StoreXs

6.2.3 Experiment Results

The results for the first experiment, in which TPC is varied, appear in Figure 6.3. It is observed as the number of threads per cluster is varied, the CLPush benchmark outperforms the NOCLPush benchmark consistently by around 3 times in terms of completion time.

The results for the second experiment, when the number of rounds is varied, appear in Figure 6.4. The relative speedup of CLPush vs NoCLPush increases from 2.6× to 2.9 times as the number of rounds increases from 10 to 10,000, plateauing by 1,000 [1]. This might be possibly be because as we increase the number of rounds, the effect of variability in timing measurements becomes less prominent.

Information gathered from other statistics also indicate that the percentage of CLPush failing (due to coherence races) in total in each experiment is also very small. For example,

[1]The speedup difference between 1000 rounds and 10000 rounds does not seem to be significantly different with the 10000 round experiment taking $>$ 10 hours, so I decided to do the third part of the experiment with 1000 rounds.

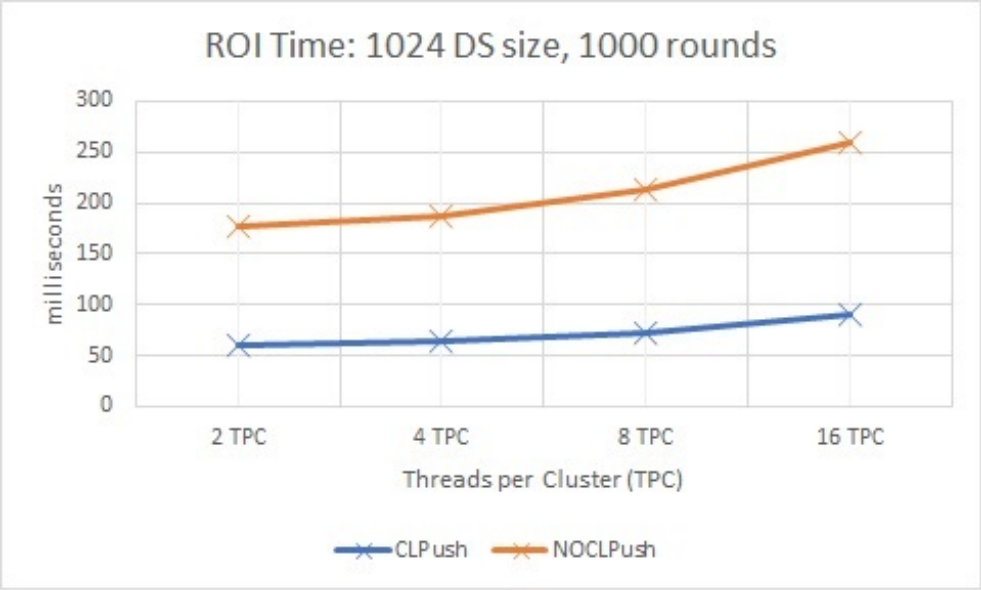

Figure 6.3: Runtime for CLPush vs NoCLPush benchmarks, varying TPC (lower is better).

only about 0.09% of CLPush failed in total for the 10K round experiment (with about half of the failures occurring at the directory and the other half at the consumer's L2 cache).

One drawback of SE mode is that it does not model the effect of other factors such as interaction with the OS, page table walks, etc. and focuses more on the user-space behavior of the program. On the upside, repeatable results are obtained.

In the third experiment (Figure 6.5), starting with an extremely small array size (12 slots), we see that the speedup of CLPush over NoCLPush is 1.8×. As the size increases up to 1024 slots, the speedup increases to 3.4× after which it plateaus. This happens possibly because increasing size of the array means a higher number of cells get successfully in the consumer L2 cache.

6.2.4 Discussion

The instruction and memory traces support my hypothesis in Section 4.4 that, as the consumer reads each slot, it will move the entire cache line for that slot over to its own L1 cache from another cluster, and this slows down the overall progress of the round (due to cross socket reads). Now, producers have to incur the cross-socket cost of updating slots in both CLPush and NoCLPush. However, CLPush saves time for the consumer when there

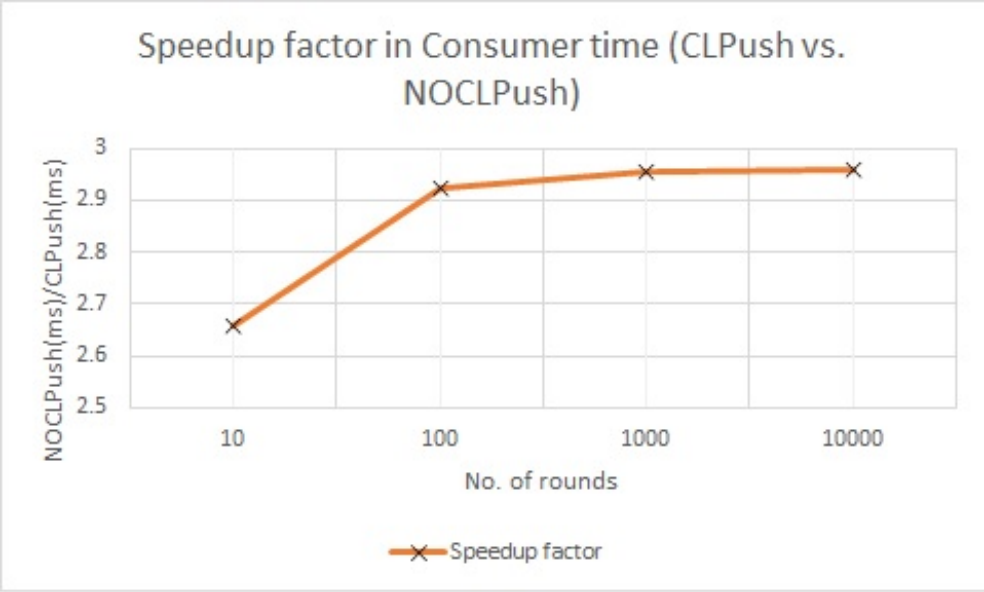

Figure 6.4: *Consumer speedup (CLPush vs. NOCLPush) vs. number of rounds*

are a lot of producers concurrently writing and pushing the value to the consumer's L2 cache. While there might be contention on the first few slots as the consumer is reading and producers are writing to the same slot, a large number of producers are writing into the cells and pushing their values over to the consumer cluster's L2, eventually outpacing the consumer. By the time the consumer reaches these *pushed* cells, it will incur a local L2 read latency as opposed to a cross-socket L1 read latency. In my implementation of CLPush, the CLPush transaction aborts i.e. gets NACKed if the destination L2 is already full. Hence, if the size of pushed data becomes large enough to fill the L2 cache, then it may possibly have no impact on the consumer side (this claim was not tested in my experiments). Local read hits will thus speed up the progress of each round, so much so that the CLPush version is consistently $\approx 3\times$ the speed compared to the base scenario.

6.2.5 Concurrent Queue Microbenchmark

Description: This microbenchmark is based on a doubly locked concurrent queue implementation by M. Michael and M. Scott [29]. The queue has two locks, one for the head and one for the tail, in order to enable concurrency between and push and pop operations (see Figure 6.6).

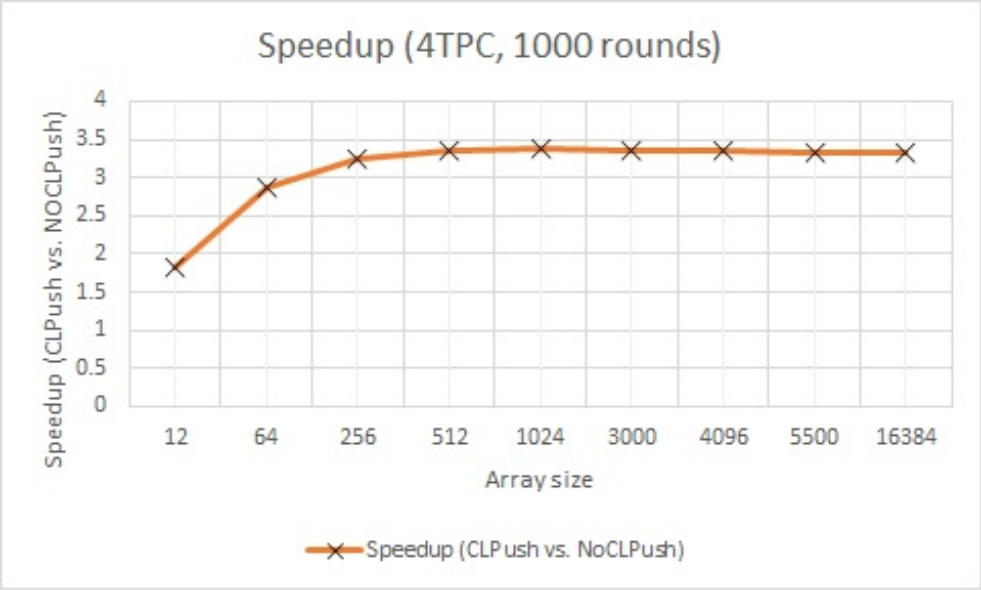

Figure 6.5: *Speedup factor vs. Data structure size (in no. of slots)*

The producer-consumer problem is tested using shared data structure as well. A noteworthy distinction between the previous microbenchmark and this one is that the previous microbenchmark models a contiguous, constant sized, statically allocated shared buffer memory. Whereas, this microbenchmark models a shared, unbounded buffer with dynamically allocated, pointer based memory which changes in size. This benchmark also involves a memory allocator, which can have a major influence on the results.

The queue supports multiple producers and consumers. As mentioned in Chapter 3, in a producer-consumer problem consumers could benefit from getting more local L2 hits when data is CLPushed.

I test this hypothesis by implementing the double lock queue in [29] and analysing performance variation between the CLPush and NOCLPush case.

In this benchmark, each producer (respectively, consumer) thread tries to push (respectively, pop), a fixed number of nodes. If any producer or consumer reaches the maximum number of operations per thread, a global flag causes all the threads to exit. The reasoning behind this is that the producers and consumers should run roughly for the same amount of time (eliminating any long tails in the distribution of thread running times).

If a consumer tries to pop and the queue is empty, it repeatedly retries until a producer pushes one into the queue and the pop succeeds. All push operations are successful. Only

Writer:

```
template<class T>
bool dlqueue<T>::push
(const unsigned int tid,T data) {
    Node<T>* node = new Node<T>(data);
    node->next = nullptr;
    tail_lock.lock();
        tail->next = node;
        #ifdef CLPUSH_ENQUEUE
        clpush(tail->next);
        #endif
        tail = node;
    tail_lock.unlock();
    return true;
}
```

Reader:

```
template<class T>
bool dlqueue<T>::pop
(const unsigned int tid, T* ref) {
    head_lock.lock();
        Node<T>* node = head;
        Node<T>* new_head = node->next;
        if(new_head == nullptr) {
            head_lock.unlock();
            return false;
        }
        *ref = new_head->_data;
        head = new_head;
    head_lock.unlock();
    delete node;
    return true;
}
```

Figure 6.6: *Double lock concurrent queue benchmark.*

successful operations are counted towards throughput (operations per second), which is the performance metric displayed in the plots.

While a detailed description of this queue implementation is given in [29], it is useful to understand how the queue implements the push and pop operations to understand how CLPush will help accelerate it.

The push and pop operations are protected by the head and tail lock respectively. The tail pointer in this queue always points to a valid node. Hence, if a time comes for the node which was pushed by a thread to be popped, the node's address will be first read into the `new_head` variable. This involves a dereference of the `node->next` field which is the same as `head->next`. Thus, CLPushing the value of `next` will potentially speed up the dereference eventually happening in the pop operation.

Pseudocode for the push and pop operation on the queue appear in Figure 6.6. The value `const unsigned int tid` denotes the thread ID of each thread which invokes `push` or `pop`. The node structure is mentioned in Figure 6.7. There are other potential places where CLPush can be used. For example, we can choose to CLPush both the `next` pointer of current tail as well as the `data` which the tail node contains. We could also potentially

Node:

```
template<class T>
class Node {
    public:
    T _data;
    volatile char padding0[64 - sizeof(T)];
    Node* next;
    volatile char padding1[64 - sizeof(Node*)];
    Node(T data): _data(data), next(nullptr) {}
};
```

Figure 6.7: *Double lock concurrent queue benchmark - node.*

choose to push the locks held by the producers or consumers. For this experiment though, I only CLPush the `next` pointer of the `tail` node (as mentioned in Figure 6.6).

6.2.6 Experiment Methodology

The queue experiment varies the number of operations (pushes and pops) producers and consumers do, and keeps the number of producers and consumers fixed.

The queue is prefilled with 1000 nodes before threads are started, in order to reduce the possibility of consumers draining the queue quickly and spin waiting on an empty queue.

The number of *operations* (pushes or pops) per thread varies in $\{1000, 10K, 100K, 1M\}$ for a 4 TPC, 2 cluster system.

Any parts of the methodology not specified are as in the previous producer-consumer microbenchmark. As in the previous experiment, this benchmark also places all the consumers in a separate socket (specifically, cluster 1). All the producers are spread across the other sockets.

6.2.7 Experiment Results

Results appear in Figure 6.8 and Figure 6.9. As the number of operations per thread varies, the throughput of producers in the CLPush benchmark is consistently lower than

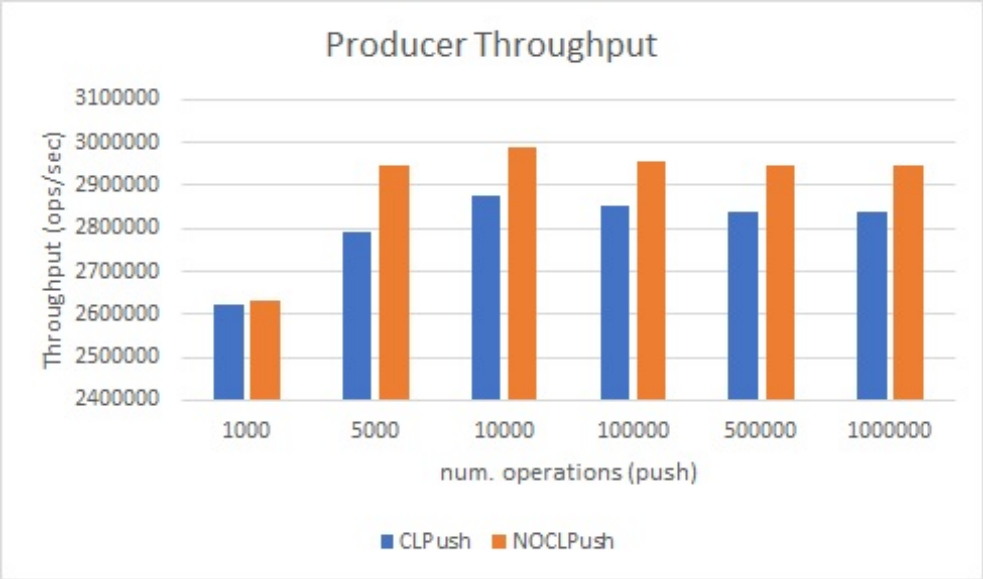

Figure 6.8: *Producer Throughput (ops/sec) vs. number of push operations*

in the NOCLPush benchmark. This is consistent with producers having to do more work to push data to a destination (in addition to storing the produced value).

On the other hand, consumers see an increase in throughput when the CLPush data is compared with the NOCLPush benchmark. An additional observation is that, as the number of operations per thread is increased, the difference in performance between the CLPush and NoCLPush benchmarks decreases.

Hypothetically, if the next pointer of a freshly enqueued node were to be CLPushed to the consumer's L2 cache, when time comes to pop this node, the consumer will find it in its local L2 cache, reducing the time needed to dereference it. This is reflected in the increase in consumer throughput with smaller numbers of operations per thread.

To explain why the improvement offered by CLPush diminishes as the number of push and pop operations per thread increases, I note the following. My current implementations of CLPush never cause cache evictions at the destination, and hence have no effect if the destination cache is full. As a result, for larger numbers of operations per thread, repeated CLPushes slowly fill up the destination cache over the course of an experimental run. Once a certain threshold is reached (essentially the capacity of L2 cache), the destination L2 will not be able to add any more data and will start to send NACKs to the directory in response to CLPush's StoreX requests. This is reflected in the StoreX statistics I gathered from the experiments. It was observed that the percentage of StoreXs failing only becomes non-zero

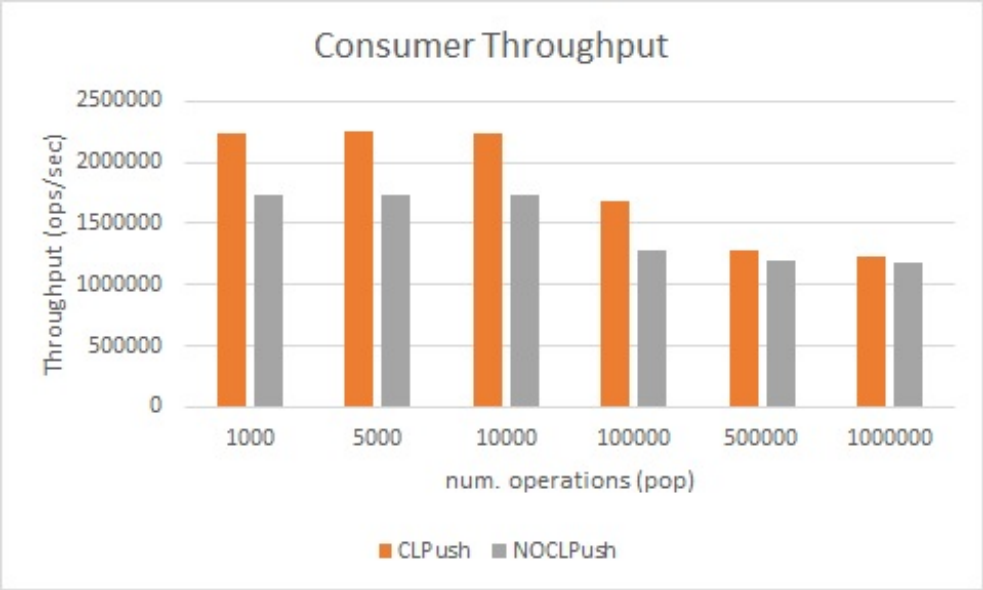

Figure 6.9: *Consumer Throughput (ops/sec) vs. number of push operations*

after a certain threshold of operations have been performed.

I emphasize that this is a result of my implementation of MOESI_coupling. In a real hardware implementation, it might be a better choice to do a cache line replacement at the destination cache (perhaps predicated on a programmer hint bit, similar to hints provided to prefetch instructions).

How does the round based producer-consumer benchmark avoid this issue? In that benchmark, the producer writes to the same shared memory array again and again in rounds, hence bringing the cache line back to the producer's L1 cache and invalidating the lines at the consumer's L2 cache (enabling subsequent CLPush instructions to take effect). In the absence of an improved implementation of CLPush in hardware, this suggests that ring buffers and other static data structures, or data structures that reclaim and reuse a relatively small amount of memory to store nodes (e.g., by using object pooling for nodes), would be a better fit for CLPush.

6.2.8 A note on experiment statistics

In practice, whether or not CLPushing a value is actually beneficial in an application can be difficult to determine.

For example, it is unclear how the choice of memory allocator affects the performance and behavior of CLPush. I used Microsoft Research's `mimalloc` [26] to reduce the usage of unnecessary system calls in the Queue benchmark. This is important because allocation and deallocation happens in every `push` and `pop` operation. Some allocators perform more system calls than others, and I wanted to keep the program running in userspace as much as possible, because it is unclear how realistic timing measurements are in Gem5's SE mode when a thread performs a system call.

Another important point to note is that Gem5, like any open-source software project, is not immune to software bugs and errors. Indeed, I also observed that SE mode also had certain possible errors with respect to statistics collection. Specifically, I encountered a bug in Gem5's statistics collection mechanism where a function which is supposed to zero out statistics after being called did not do so. I reached out to one of the Gem5 community contributors and they suggested a one-line fix for it.

Additionally, Gem5 gathers statistics over the entirety of a program execution, including experiment setup and tear-down, so relying solely on its counters would introduce intolerable error in some of my experiments. Hence, I relied mainly on the read timestamp counter (RDTSC) instruction, and RDTSCP, which together provide a low overhead way to read timestamps from the processor that are linearly related to wall clock time (and are also supported by Gem5 on the x86 processor I used). This helped in gaining a rough understanding of whether CLPush was causing speedup in the region-of-interest.

To reduce uncertainty in the results, I also tried to change small parts of code in a test application to determine how these changes affect the statistics gathered in Gem5. This method gave me insights into which statistics to trust and which not to trust. For example, I could reasonably trust any statistics gathered related to CLPush, as the CLPush instruction is not called anywhere other than the test application, as it is an artificial instruction which is highly unlikely to be used in any program setup code for SE mode.

Chapter 7

Program Artifacts

In this chapter, I will examine the general program behavior using CLPush as an instruction, and how it interacts with memory objects. This chapter mainly looks at the potential use cases, interesting behaviors and other miscellany.

The first interesting behavior I found was while writing up a basic test application to test the usage of the CLPushed variable. This behavior comes up when CLPushing a stack variable which has been declared close to other stack variables.

If the stack variables are not padded correctly, then any store of the adjacent variables results in a negative interaction of the CLPush instruction with adjacent instructions, resulting in a kind of false sharing. Consider the following scenario:

```
int shared_val;
int j;
CLPush(shared_val);
j = 0xabababab;
```

The subsequent write after CLPush will bring the ownership of the whole cache line back to the CLPushing thread before the remote thread has the chance to read the pushed data. This can happen on heap as well if malloc results in two variables occupying the same cacheline.

While it does not make much sense for a stack variable to be CLPushed to any other core explicitly, CLPushing a stack variable might be of some use in the domain of User Level Threading or when threads share stacks.

Code:

```
clpush(shared_val);
printf("printed\n");
```

Figure 7.1: *Calling printf after CLPush*

Disassembly (O0):

```
repnz mov -0xc(%rbp),%ecx
movabs $0x400774,%rdi
mov    $0x0,%al
callq  0x400410 <printf@plt>
```

Disassembly (O1):

```
mov    $0x400764,%edi
callq  0x400410 <puts@plt>
xor    %edi,%edi
```

Figure 7.2: *Printf disassembly with -O1.*

The second interesting behavior that I found was that when the test program was compiled with a low optimisation level (eg. -O0), the compiler calls printf in Figure 7.1 (Figure 7.2 O0 disassembly). Whereas, on more aggressive optimisation, this changes into a call to `puts` ((Figure 7.2 O1 disassembly)).

It was observed from the traces that the `printf()` call causes the return IP to be spilled onto the stack, on the same line as the CLPushed variable. This implies that it is a write on to the cache line containing the CLPushed variable `shared_val`, defeating the whole purpose of CLPush.

This is different from the `puts` case where it was observed that the return address is stored on a cache line separate to that where the CLPushed variable resides.

Chapter 8

Future Work and Directions

Future implementations of CLPush can be augmented with a number of features. Currently, for the implementations in Section 5.1.2, Section 5.2.1 and Section 5.2.2, the issuer can only push a cache line onto a hard-coded destination in the system. Instead, there can be a version where the CPUID(s) is passed along with the data to push,

```
clpush(data, cpuid);
```

This will hint the cache from where CLPush transaction is initiated to push the data to the core specified in the `cpuid`.

Another potential optimisation that could be implemented makes use of the use case to store a value and immediately CLPush it. Currently stores and clpush instructions have to be executed one after the other as separate instructions. Instead, we can have better integration of stores with CLPush to combine them into one instruction that will have the option of just sending the existing data in a variable, or writing a new value to the variable and CLPushing the fresh value.

One more extension to CLPush can transfer lines which are in other states like Owned or Shared.

Next, in the current version of the instruction, if the destination cache is currently full (or, more precisely, the destination cache associativity set), CLPush does not trigger any replacements due to the possibility of complicated race conditions arising from the replacement transaction interfering with progress of CLPush. A possible solution to this might be to add a *victim cache* for CLPushed data, reducing contention between the destination cache and CLPushed data.

On real systems, CLPushing too often may lead to flooding of the network with too many messages. (The same thing can happen with prefetching instructions.) An approach to solving issues arising from large amounts of data transfer through the directory, is that the CLPush protocol can be made to support direct cache to cache transfer, with coherence supported accordingly.

Chapter 9

Conclusion

In this thesis, I contribute a novel cache manipulation instruction called CLPush. I show how CLPush can be integrated in different coherence protocols, namely MESI and MOESI.

I also show how effectively Gem5 allows users to prototype a new instruction and analyse its effects. Using Gem5's custom testers and its purpose-built coherence modelling infrastructure like Ruby enables both computer architects and software programmers to find a common ground and flesh out their optimisations and ideas.

I show how producer-consumer applications stand to benefit from this instruction. It is important to note that the design of CLPush can greatly affect the kinds of applications it can speed up. Since the cache line is pushed to other cache(s) in the shared state, read heavy workloads stand to benefit from this version of CLPush. The writer threads may incur some extra overhead at the cost of CLPushing and making the written data readily available on the destination cache. On the other hand, if CLPush were to be designed in such a way that it does *ownership transfer*, a different set of applications such as writer-writer workloads may benefit from this. All in all, the applications which could have speedup depend largely upon the semantics and implementation of CLPush.

In an era of increasing non-uniformity and heterogeneity it is becoming increasingly important for systems programmers to be aware of data placement. The new CLPush instruction directly addresses this need, giving programmers the ability to leverage knowledge about access patterns to improve cache locality—the dominant performance factor in modern multicore applications.

References

[1] biglittle. https://en.wikipedia.org/wiki/ARM_big.LITTLE. Accessed: 2023-07-14.

[2] chi-manual. https://developer.arm.com/documentation/ihi0050/latest/. Accessed: 2023-07-14.

[3] chi-stashing. https://developer.arm.com/documentation/102407/0100/Cache-stashing. Accessed: 2023-07-14.

[4] Mesi protocol. https://en.wikipedia.org/wiki/MESI_protocol. Accessed: 2023-07-14.

[5] Moesi protocol. https://en.wikipedia.org/wiki/MOESI_protocols. Accessed: 2023-07-14.

[6] Moesi state machine. https://www.gem5.org/documentation/general_docs/ruby/MOESI_CMP_directory. Accessed: 2023-07-14.

[7] Proximity coherence for chip-multiprocessors. https://doi.org/10.17863/CAM.16370.

[8] Proximity coherence for chip-multiprocessors - phd thesis. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-810.pdf.

[9] X86 decoding. https://wiki.osdev.org/X86-64_Instruction_Encoding. Accessed: 2023-07-14.

[10] Brian Klug Anand Lal Shimpi. s4-bug. https://www.anandtech.com/show/7164/samsung-exynos-5-octa-5420-switches-back-to-arm-gpu. Accessed: 2023-07-14.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[12] Liqun Cheng and John B. Carter. Extending cc-numa systems to support write update optimizations. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[13] James Goodman Chinya Ravishankar. bus-snooping. https://www.cs.ucr.edu/~ravi/Papers/NWConf/ravishankar_83.pdf. Accessed: 2023-07-14.

[14] Felix Cloutier. Cldemote. https://www.felixcloutier.com/x86/cldemote. Accessed:.

[15] Felix Cloutier. Clflush. https://www.felixcloutier.com/x86/clflush. Accessed: 2010-09-30.

[16] Felix Cloutier. Clwb. https://www.felixcloutier.com/x86/clwb. Accessed: 2010-09-30.

[17] Felix Cloutier. Intel mesif. https://www.realworldtech.com/common-system-interface/5/. Accessed:.

[18] Andrei Frumusanu. Xeon 3830 access latencies. https://www.anandtech.com/show/16594/intel-3rd-gen-xeon-scalable-review/4. Accessed: 2023-07-14.

[19] Håkan Grahn, Per Stenström, and Michel Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 11(3):247–271, 1995.

[20] Patrick N. Conway John D. McCalpin. Pushsh. https://patentimages.storage.googleapis.com/92/f1/c4/7d2a80f82a40fb/US8099557.pdf. Accessed: 2010-09-30.

[21] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[22] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGARCH Comput. Archit. News*, 30(5):211–222, oct 2002.

[23] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 211–222, New York, NY, USA, 2002. Association for Computing Machinery.

[24] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGOPS Oper. Syst. Rev.*, 36(5):211–222, oct 2002.

[25] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGPLAN Not.*, 37(10):211–222, oct 2002.

[26] Daan Leijen, Zorn Ben, and Leo de Moura. Mimalloc: Free list sharding in action. programming languages and systems, 11893 (2019), 2019.

[27] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, page 148–159, New York, NY, USA, 1990. Association for Computing Machinery.

[28] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *SIGARCH Comput. Archit. News*, 18(2SI):148–159, may 1990.

[29] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 267–275, New York, NY, USA, 1996. Association for Computing Machinery.

[30] Mark Hill Vijay Nagarajan, Daniel Sorin and David Wood. *LaTeX — A Primer on Memory Consistency and Cache Coherence*. Morgan-Claypool, second edition, 2020.

[31] D.A. Wood, G.A. Gibson, and R.H. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Design & Test of Computers*, 7(4):13–25, 1990.