Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Erling Feet Nesset

Improving the first-level cache bandwidth in the Berkeley Out-of-Order Machine

Master's thesis in Computer Science
Supervisor: Magnus Jahre
Co-supervisor: Björn Gottschall
June 2023

NTNU
Norwegian University of
Science and Technology

Erling Feet Nesset

Improving the first-level cache bandwidth in the Berkeley Out-of-Order Machine

Master's thesis in Computer Science
Supervisor: Magnus Jahre
Co-supervisor: Björn Gottschall
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Problem Description

FireSim is the state-of-the-art FPGA-accelerated cycle-exact simulator and typically uses the Berkeley-Out-of-Order Machine (BOOM) when configured to evaluate high-performance computer architectures. Unfortunately, BOOM's memory system provides sub-optimal bandwidth for some applications. More specifically, the L1 cache can provide sub-optimal bandwidth for eviction-heavy applications because the time the cache spends handling evictions grows linearly with the number of Miss Status Holding Registers (MSHRs). This limits the amount of Memory Level Parallelism (MLP) that the BOOM can exploit — thereby yielding sub-optimal performance for memory-sensitive applications. The objective of this master thesis is to investigate approaches for addressing BOOM's L1 cache bandwidth problem. The student should use the analysis performed during the autumn project to identify optimization opportunities and then implement and evaluate key optimizations in FireSim. The evaluation should focus on the streaming microbenchmark used during the autumn project. If time permits, the student should evaluate key optimizations on complete benchmarks (e.g., SPEC CPU2017).

Abstract

As processors have hit the memory wall, they have used memory-level-parallelism (MLP) to hide memory latency. To exploit MLP, processors need sufficient memory bandwidth and as such bandwidth has become integral to processor performance. I thoroughly analyze the bandwidth the Berkeley Out-of-Order Machine (BOOM) can achieve to different levels of its memory hierarchy. By comparing this bandwidth to what should be theoretically possible, I identify a bottleneck in the L1 cache's writeback handling. This bottleneck significantly reduces bandwidth for eviction-heavy applications. I optimise the writeback handling by pipelining the writeback unit, enabling it to read dirty cache lines from the cache while writing back another cache line. This improves write bandwidth by 30% to the L2 and 20% to the main memory. This increase in bandwidth reduces the execution time of the LBM benchmark with software prefetching by 3,7% for under a 0.2% increase in resource consumption. In addition, I discovered a deadlock in the BOOM caused by the L2 not acknowledging a writeback when multiple new cache lines are requested while dirty cache lines are written back. The deadlock is avoided by not requesting new cache lines when dirty cache lines are being written back.

Sammendrag

Ettersom moderne prosessorer de siste tiårene har truffet minnegapet, har de brukt minne-nivå-parallelisme(MLP) for skjule forskjellen i ytelse mellom prosessoren og minnet. For å utnytte MLP trenger prosessorer nok båndbredde og båndbredde har derfor blitt viktig for ytelsen til prosessorer. I denne oppgaven analyserer jeg minnebåndbredden Berkeley Out-of-Order Machine(BOOM)-kjernen klarer å oppnå til forskjellige deler av minnehierarkiet. Ved å sammenligne denne båndbredden med hva BOOM-en teoretisk sett bør kunne oppnå identifiserer jeg en flaskehals i nivå 1 hurtigbufferens tilbakeskriving av skitne hurtigbufferlinjer. Denne flaskehalsen reduserer minnebåndbredden til programmer med mye tilbakeskriving betraktelig. Jeg optimaliserer tilbakeskrivingen ved å pipeline enheten som håndterer tilbakeskriving, slik at den kan lese ut en skitten linje fra cachen samtidig som den skriver tilbake en annen linje. Dette øker båndbredden til nivå 2 hurtigbufferen med 30% og båndbredden til hovedminnet med 20%. Denne optimaliseringen øker ytelsen til LBM med prefetching i programvare med 3,7% for under 0.2% økning i ressursbruk. I tillegg oppdager jeg en vranglås i BOOMen forårsaket av at L2 hurtigbufferen ikke bekrefeter en tilbakeskrevet skitten hurtigbufferlinje når flere hurtigbufferlinjer blir forespurt mens den skitne linjen blir skrevet tilbake. Vranglåsen unngås ved å ikke forespørre nye hurtigbufferlinjer mens en linje blir skrevet tilbake.

Preface

This master thesis is written as part of TDT4900 and focuses on improving the cache bandwidth of the BOOM's L1 cache. The master thesis builds on my previous work done during the project thesis [1] performed in the autumn of 2022. In my project thesis, I investigated the memory bandwidth of the BOOM when exploiting various amounts of MLP. As the project thesis described caches and the BOOM's memory system in detail there is some overlap between my autumn project in sections 2 and 3. In addition, I build on the insights from my project thesis for the initial analysis in section 6 as suggested by the problem description.

I would like to thank my supervisor Magnus Jahre and my Co-Supervisor Björn Gottschall for their continued support throughout the semester. They have both been of tremendous assistance during the entire semester and have kept me motivated even when I endured difficulties and had to overcome challenges.

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Motivation

Computer architecture simulators are the main approach for evaluating new architectural changes [2–5]. They are useful as they offer a low-cost and simple way to test new changes without physically implementing architectural changes. However, as computer architectures have grown more complex, the software used to evaluate performance has grown increasingly demanding [6]. This poses a problem for software simulators, as the time needed to execute cycle-accurate simulations benchmarks such as SPEC2017 increases drastically [5, 6].

FireSim [7] is the state-of-the-art FPGA-accelerated cycle-exact simulator. It can drastically reduce the simulation time of workloads compared to software simulators [5, 7] while still being cycle accurate. This is done using FPGAs to achieve magnitude higher performance than simulating the same architecture in software [7]. For simulators to be useful, the architectures they simulate must be designed so as to not have sub-optimal implementations that can cause bottlenecks. This is because new architectural changes tested on simulators may have their results skewed by said bottlenecks, preventing proper evaluation.

When configured to simulate an out-of-order it uses a configuration of the Berkeley Out-Order-Machine (BOOM) [8] generator. As a BOOM core can be generated with different parameters such as issue width, cache size and entries in the ROB, performance varies depending on which configuration is being used with FireSim. As with other modern high-performance processors, the BOOM achieves high performance by executing multiple instructions in parallel, thereby achieving i high degree of Instruction-Level-Parallelism(ILP). This ILP is achieved by executing instructions out of program order in order to extract the most available ILP from a program.

Figure 1.1 shows that the latency of memory instructions has diverged from DRAM latency in the last decades. This has given rise to the memory wall [10]. To overcome this challenge processors have been forced to utilise the increased bandwidth offered by memory systems [11] to hide the latency of memory accesses by exploiting Memory-Level-Parallelism(MLP) [12]. MLP is informally how many

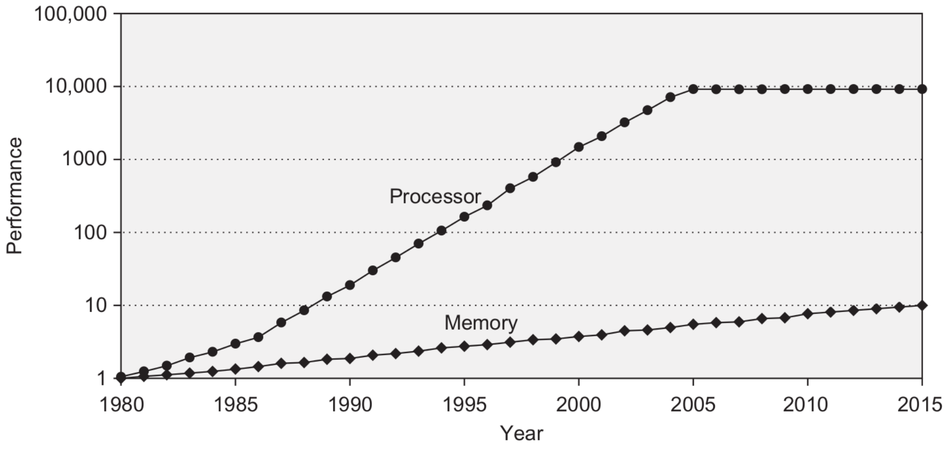


Figure 1.1: Latency of processor memory instructions compared with DRAM latency since 1980. Taken from Computer Architecture and Design 6th Edition [9].

memory instructions that can be kept in flight at a time. As the latency of memory instructions has increased relative to processor speed, achieving a high MLP has become instrumental in effectively hiding the latency of memory instructions. Because data from memory is needed for almost all other instructions, effectively hiding memory latency has become imperative for achieving high performance in modern processors. Therefore, memory bandwidth is highly dependent on exploiting MLP and is important for achieving ILP in general.

To facilitate a high degree of memory level parallelism and to reduce the mentioned increased latency in memory systems, several levels of caches have been introduced. Caches exploit temporal and spatial locality by offering fast hardware-controlled buffer storage for data that will likely be reused in the future, thereby reducing average access latency. In addition, they have also been designed to support MLP by being able to handle several misses and requests at a time. Therefore, how caches support MLP is integral to how much of it can be exploited by the processor. If a low-level cache is not able to support a large amount of MLP, it will become the bottleneck of the entire system, as the latency will increase and the processor is unable to properly hide the latency of memory instructions by exploiting MLP.

As exploiting MLP and for that delivering sufficient bandwidth between the CPU and the memory system is necessary to achieve high performance, it is important that a simulator does not have any bottlenecks in the memory hierarchy caused by sub-optimal implementation. This could lead to changes implemented in a simulated architecture having a different impact on real-world systems with a better memory hierarchy. An example would be a change implementing the usage of functional units, which could have a lower impact on the performance of a system with insufficient memory bandwidth and exploitation of MLP.

1.2 Interpretation of Problem Description

The project description states that the goal of this thesis is to address the bandwidth problem the BOOM experiences with eviction-heavy applications discovered during my autumn project [1]. More specifically, applications with many cache misses that result in the writeback of dirty data suffer from sup-optimal bandwidth when the BOOM has many Miss Status Holding Registers. Using the analysis in the autumn project, I was to investigate approaches to alleviate this bottleneck. Subsequent optimisations should then be evaluated by utilising FireSim to simulate the streaming microbenchmark and, if time permits, complete benchmarks such as SPEC CPU2017.

As the streaming microbenchmarks measure the bandwidth achieved at different levels of the memory hierarchy, the bandwidth must be measured without any optimisations before any optimisations can be evaluated, since any optimisations must be compared with the previously measured bandwidth for evaluation. In addition, the theoretical limit of the bandwidth, i. e. the maximal amount of data which can be transferred to and from memory each second must be calculated and compared with the measured bandwidth to see if further improvement is possible. While the theoretical limit, i.e. the maximal amount of data that can be transferred over the connection between the core and the memory hierarchy is not achievable, it is possible to get close. Therefore, it is useful to evaluate where it is possible to improve bandwidth. Based on this, I define the following tasks from the problem description:

1. Measure bandwidth and calculate the theoretical limit
2. Identify and improve bottlenecks
3. Measure bandwidth with optimisations
4. Evaluate optimisation on complete benchmarks

1.3 Contributions and Outline

In this thesis, I first use several microbenchmarks to measure the bandwidth both to and from the BOOM core itself and to and from the L1 cache. By doing this, I address task 1 and replicate my findings from my autumn thesis on a larger BOOM-config, showing that the BOOM delivers insufficient bandwidth for writeback-heavy applications. More specifically, the BOOM encounters a problem where the writeback of dirty cache lines becomes the main bottleneck when handling a large number of concurrent misses for writeback-heavy applications, limiting the amount of MLP the BOOM can exploit and thereby the bandwidth it can achieve to memory.

Figure 1.2 shows the memory hierarchy of a high-performance configuration of the BOOM, as well as how the miss handling of the L1 cache interacts with the L2 cache. As the BOOM is configurable, the number of MSHRs and therefore the number of misses that can be handled concurrently varies between different

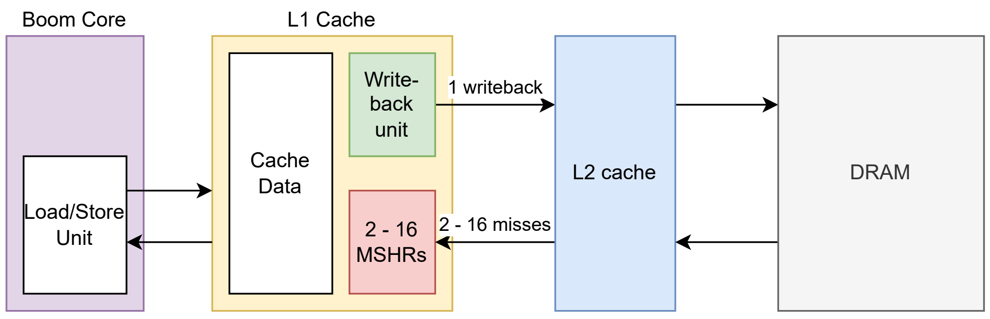


Figure 1.2: An overview of the BOOM's memory system.

configurations of the BOOM. Regardless of BOOM configuration, all memory requests must first pass through the L1 cache. Any misses will then be sent to the L2 and thereafter DRAM. As all misses must pass through the L1, it may become a bottleneck if it cannot deliver as high bandwidth as the L2. It delivers increased bandwidth by increasing the number of in-flight misses. However, any miss that results in the eviction of a dirty cache line must first write that line back to the L2 cache. This is handled by the writeback unit, yet regardless of the number of misses being handled concurrently, the writeback unit can only handle one writeback at a time. As the number of misses being handled concurrently rises, the writeback unit becomes a bigger and bigger bottleneck, until most of the time spent resolving an L1 miss is spent waiting for the writeback unit to become available.

To address task 2, I contribute a pipelined writeback unit that is able to reduce the time spent waiting on a writeback significantly. The pipelined writeback unit increases the write bandwidth by 20% for main memory and 30% for the l2 cache, addressing task 3. Furthermore, I demonstrate the usefulness of increased bandwidth by showing the execution time of LBM with software prefetching is reduced by up to 4%, while the pipelined writeback unit increases resource consumption by under 0,2%, and by this address task 4. While evaluating the pipelined writeback unit I encounter a deadlock in the BOOM's implementation of the Tilelink protocol. This deadlock prevented Linux from booting and was necessary to avoid in order to evaluate the performance of my optimisations. The deadlock is caused by the L2 cache not acknowledging a writeback when multiple cache lines are requested while the writeback is in progress. The deadlock is avoided by not requesting new cache lines while a cache line is being written back to the L2 cache.

1.4 Outline

- In Chapter 2 I explain the theoretical background relevant to the BOOM and its memory system addressing.
- In Chapter 3 I describe the data cache and miss handling of the BOOM, a part of the BOOM that is currently not documented anywhere.

- Chapter 4 contains a brief explanation of the Tilelink protocol, as well as documenting how I discovered and avoided a deadlock within the BOOM's Tilelink implementation.
- In Chapter 5 I present the experimental setup, explain why the benchmarks used were chosen and how they were used. In addition, I present the configuration of the BOOM-core used.
- In Chapter 6 I measure the bandwidth and compare it to what is theoretically possible, as well as identify bottlenecks explaining the difference between the two.
- In Chapter 7, I present optimisations to alleviate the bottlenecks and evaluate their effect on bandwidth, resource usage and impact on the performance of LBM with and without software prefetching.

Chapter 2

Background

2.1 Out-of-order processing units

2.1.1 Out-of-order execution

The goal of high-performance processors has been to execute as many instructions as possible per second, thereby executing programs as fast as possible. To increase the number of instructions executed, either the number of instructions executed per cycle (IPC) or the clock cycle speed must be increased. As most high-performance processors will have more than one functional unit, such as an ALU or FPU, one way of increasing IPC is to use multiple functional units at the same time. This is referred to as *Super scalar multiprocessors* [13] which are capable of achieving an IPC > 1.

However, to be able to use multiple units, program instructions must be able to be executed in parallel. If there are dependencies between instructions, such as where the next instruction needs the result of the previous one not yet executed, the next instruction must wait for the first one to be executed. For example, consider the following instructions shown in figure 2.1. As there are dependencies between instructions 1, 2 and 3, they cannot have overlapped execution. On the other hand, there are no dependencies between instructions 4, 5 and the rest. Therefore if instructions can be executed out of program order, the executions of instructions 1, 4, and 5 can be overlapped achieving given enough functional units. As multiplication may take more than one cycle, overlapping the execution of instructions 1, 4 and 5 may successfully hide the latency of the multiplication.

Out-of-order processors emerged as a way of making use of multiple functional units [15]. This is achieved by exploiting *instruction-level-parallelism (ILP)*, or instructions that are able to be executed in parallel. Out-of-order execution is able to exploit more ILP than traditional in-order execution by executing instructions without dependencies regardless of whether they are behind other instructions with dependencies in program order. As more instructions can be considered for parallel execution, it is easier to execute as many instructions as possible in parallel.

Instruction	In-order	Out-of-order
(1) add r2, r0, r1	•	•
(2) add r1, r3, r2		
(3) add r2, r1, r2		
(4) add r3, r3, r4		•
(5) mul r4, r4, r4		•

Figure 2.1: Assembly code illustrating the benefits of Out-of-Order execution, marking which instructions can have overlapped executions in an in-order and out-of-order core. Inspired by Carlson et. al. [14].

Additional ILP can be achieved by speculatively executing instructions before we know if a preceding branch is taken or not. This allows a processor to continue executing instructions and keep IPC high even when having to wait for branches to be evaluated. However, this comes with the challenge of having to guess which instructions will be executed after a branch without knowing if the branch will be taken or not. In addition, if the guess was wrong the processor has to undo the instructions it guessed would be executed.

As the resources available to computer architects increased, out-of-order become more widespread [16]. Today, most modern high-performance processors use some form of Out-of-order execution, meaning that it is not only relevant for large computer clusters but also desktops, laptops and even mobile devices.

2.1.2 Components of an Out-of-order execution processor

There are many ways of implementing out-of-order execution, yet there are many common elements [17]. An example of a detailed implementation is the [16], which has been used as an inspiration for more modern open-source Out-of-order processors such as the BOOM [8]. While there are several approaches to solving some problems in Out-of-order processors, such as either explicit vs explicit register renaming, they both solve the same problem and the pipeline steps remain broadly the same.

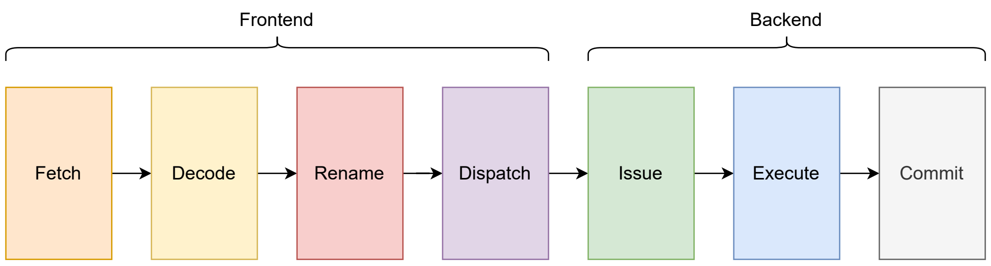


Figure 2.2: An overview of the stages in Out-of-order Execution, inspired by Palacharla et. al. [17].

A brief overview of the components in an out-of-order processor is shown in figure 2.2. The first four stages related to fetching and decoding the instructions are called the *Frontend* and are generally done in program order. The *Backend or Execution* section is what is done out-of-order. A brief explanation of each stage is given below:

- Fetch: Get instructions from Instruction cache
- Decode: Determine the type of instructions as well as operands.
- Rename: Rename registers to avoid hazards
- Dispatch: Instructions are dispatched into the Re-Order Buffer(ROB) and issue queues
- Issue: Determine when an instruction can be sent to an execution unit, and send it when ready
- Execute: The execution units of the instructions. Includes the use of ALUs, floating point units, address-generating units and the interfaces to the data cache among others. What is done varies greatly between instructions, from simply moving data between registers to executing complex floating point divisions and fetching data from memory.
- Commit: Once the previous instructions have been committed and an instruction has been executed, the instruction is removed from the ROB and the logical register state is updated.

To provide an example of how the different components of an out-of-order processor interacts, consider an example of an instructions journey through the pipeline, from fetch and decode to eventual commit. For simplicity, we consider an instruction which adds two registers together and stores the result in a third.

The instruction begins its journey in the *fetch* stage by being fetched from the Instruction cache. Some, but not all, processors fetch multiple instructions per cycle. Once the instruction has been fetched it is decoded in the aptly named *decode* stage. Here it is determined which registers are the sources and destination of our instruction, as well as that it is an add instruction. Resources needed by an instruction in later phases, such as a space in the ROB and in the issue queues are also typically allocated in the decode stage. If either the ROB or the relevant issue queue is full, the Frontend will stall until resources become available. The next stage is *rename*, where any write-after-write or write-after-read hazards between our instruction and others are resolved using register renaming. In the subsequent *dispatch* stage, the instruction information is put into the correct issue queue and the ROB buffer.

Once the instruction has left the Frontend, it reaches the *issue* stage and its journey will vary depending on the other instructions in the pipeline. If its operands are ready and an execution unit is free, the instruction will be executed at once. However, if its operands are not ready it will stay in the issue queue until it is ready to be executed. Once it is issued, i. e. sent to the execution unit, it reaches the *execute* stage. As our instruction is a simple add, it usually spends only a single cycle in this stage. However, for more complex operations such as floating point multiplication, this may not be the case. Once the instruction is executed, its result will be stored temporarily until it reaches the head of the ROB, meaning all preceding instructions in the program have been committed. Once that is the case, the instruction will reach the *commit* stage. Now the destination register will be updated with the result, and the instruction will leave the ROB as it is no longer possible for the instruction to be squashed in case of exceptions or a failed branch prediction.

2.1.3 Balance in out-of-order cores

The goal of an out-of-order core is to extract ILP. This entails that it should try to dispatch as many instructions per cycle as possible, and the different stages should be designed to support this. More formally Eyerman et. al. [18] defines a balanced Out-of-Order core as a core that can sustain a performance of D instructions per cycle in the absence of misses, where D is the maximum number of instructions dispatched each cycle. Furthermore, Eyerman's definition entails that none of the pipeline resources can be reduced without the performance being reduced. As such, a balanced core is a minimal core able to sustain an effective throughput of D instructions per cycle.

To achieve a throughput equal to the dispatch width while still having a minimal core, the resources used by the different stages of an Out-of-order core must be scaled appropriately [18]. There is a linear relationship between issue width, commit width and the number of functional units. On the other hand, the ROB size scales quadratically with the issue width, while the number of registers, issue buffer size, and load/store queues scale linearly with the ROB. Moreover, the rest of the memory system must be scaled linearly with the size of the load/store queues to support MLP, as explained in the next section.

2.1.4 Memory level parallelism

When out-of-order processors were first developed, the goal was as mentioned to use as many functional units at a time as possible. However, since then processor speed has outpaced memory speed, placing an increased emphasis on getting data from memory for keeping up ILP [10]. This is especially true for memory-intensive applications which contain many memory instructions that result in cache misses.

One way of mitigating the memory wall is exploiting Memory-level-parallelism or *MLP*. MLP is defined as the number of memory requests in flight at a current moment [12]. By overlapping memory requests, their individual latency is hidden, resulting in reduced total latency. Figure 2.3 illustrates the benefits of MLP with two examples. In subfigure 2.3a the loops are independent and can their execution can therefore be overlapped, whereas the loads in subfigure 2.3b cannot. If each load results in a cache miss with a fixed latency of 20 cycles, the loop in subfigure 2.3a can be executed in only 22 cycles given an issue width of 1 and 1-cycle additions. The loop in subfigure 2.3b has to wait for both instructions 1 and 2 to

```
  mlp-loop:
(1)  ld r2, 0(r0)
(2)  add r2, 2, r2
(3)  ld r3, 8(r0)
(4)  add r3, 2, r3
(5)  add r0, 8, r0
(6)  bne r4, r0, mlp-loop
```

(a) Loop with MLP.

```
non-mlp-loop:
  ld r2, 0(r0)
  add r2, 2, r2
  ld r3, 8(r2)
  add r3, 2, r3
  add r0, 8, r0
  bne r4, r0, non-mlp-loop
```

(b) Loop *without* MLP.

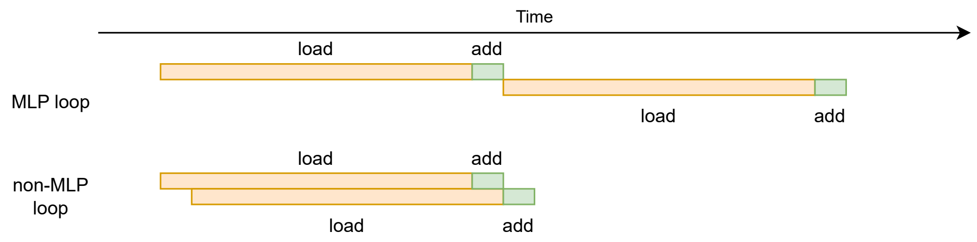


Figure 2.3: Two loops, one with MLP and one without and the execution time broken down by instruction. Execution time for instructions 5 and 6 are omitted for brevity. Inspired by Chou et. al. [12].

finish before firing the second load. This results in an execution time of 42 cycles, or roughly twice as long because the loads could not be overlapped.

As shown in figure 2.3, the processor needs to identify memory instructions that can execute in parallel. In fact, by exploiting just MLP and not ILP in general, processors achieve a lot of the speedup given by full Out-of-order execution [14]. To exploit MLP a computer architecture must support several concurrent memory instructions, either through traditional out-of-order execution or by just MLP support as described by Carlson et. al. [14]. In addition, caches must be expanded to handle the increased number of in-flight misses [19].

2.2 Caches

2.2.1 Introduction to caches

Processor speed has for a long time exceeded the access speed of DRAM main memories [10]. This has necessitated the development of smaller cache memories that have a far shorter access speed than DRAM [20]. As some caches are able to fit onto the same chip as the processor itself, their speed is able to keep up with the speed of the processor [10]. Caches reduce the latency of both instruction- and data fetching to the CPU [20], and they are instrumental in keeping up the IPC of modern processors as the gap between processor and DRAM speed has increased [10].

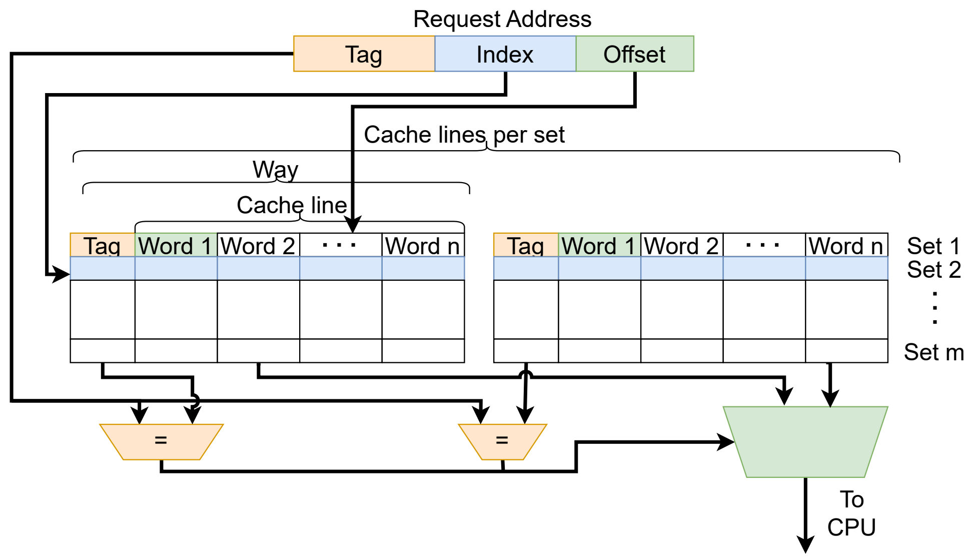


Figure 2.4: The internal organisation of a typical cache, taken from my Project thesis [1].

Cache memories work by temporarily storing a limited number of memory addresses that have either been used in the past or are believed to be used in the future. Whenever the processor wants to read from an address in memory, the cache will check if it currently stores that address. If the address is present in the cache, it will send the requested data to the processor [20]. In the other case, when the address is *not* present in the cache, the cache will request the data at that address from memory. This highlights two important terms when discussing caches:

- **Cache hit**: When the requested data is present in the cache
- **Cache miss**: When the requested data is *not* present in the cache

The ratio of these two cases for memory requests is very important for the performance of a program, as the memory latency varies greatly between them [10]. Cache memories rely on the tendency of spatial and temporal locality within memory accesses to increase the number of cache hits for programs, and by extension the average memory latency.

2.2.2 Cache organisation

Figure 2.4 shows the internal organisation of a cache and how it uses the address in a memory request to determine where in the cache the data may be located. In addition, figure 2.4 shows the process of checking whether the requested address is actually present within the cache, i. e. whether the request is a cache miss or a cache hit.

To identify individual data words within blocks and check if an address is present in the cache, the address is split into a tag, an index and a block offset. The offset is used to identify which word within a cache line the requested address refers to. The index is used to define the position in the cache a cache line may occupy. Lastly, the tag of every cache line is stored alongside the cache line within the cache, and is compared to the tag part of the request address to check if the requested address is present within the cache, as shown in figure 2.4.

The simplest caches are organised in a way there each memory location can only be stored in one place in the cache. This results in a simple cache implementation, as only a single tag comparison is needed to check whether an address is present. However, this organisation does not always interact favourably with program behaviour. If a program frequently accesses two addresses mapped to the same place in the cache in an interleaved fashion, many accesses will be misses. Therefore, most caches use some form of set-associative mapping [20].

The cache organisation shown in figure 2.4 shows a set-associative cache. Here, each cache line is mapped to one of many *sets*. The number of cache lines per set is defined as the number of *ways* in a set associative cache. Therefore, the number of *ways* in a cache dictates how many places a certain address may be placed within the cache. The cache shown in figure 2.4 is a 2-way set associative cache, meaning it has two ways.

Whenever a cache is not direct-mapped and a cache line is being fetched, a replacement policy is needed to decide which of the cache blocks present must be evicted. There are several schemes which aim to evict the cache line that will result in the lowest amount of cache misses. However, as more complex replacement policies require more complex logic, they must be weighed against other schemes to reduce misses.

As the goal of cache organisation is to primarily reduce the number of misses [20], it is important to categorise misses and how each category can be dealt with. Misses are usually organised into the following categories:

- **Capacity misses**: Misses where the memory location accessed was previously accessed, but was later removed from the cache due to the cache being full.
- **Conflict misses**: Similar to capacity misses, yet the line was evicted due to just the set being full and not the entire cache
- **Cold misses:** The first access to a location in memory made by a program. These misses will occur even with a cache with infinite capacity and where the entire cache is one set.

Capacity misses can be mitigated by increasing the capacity of the cache, while conflict misses can be mitigated by increasing the capacity of the sets, i. e. increasing associativity. However, both associativity and cache size impact the access latency of the cache itself [21]. As both cache size and associativity generally increase latency, there are limits to how large and associative caches can be while still having the desired access latency.

2.2.3 Write handling

There are in general two types of ways to interact with memory, namely read or write requests. These correspond to the various types of load and store instructions found in RISC architectures. Caches need to process both, and both need to be handled differently. Both types of requests can either be hits or misses. This gives four situations that caches need to handle:

- **Read hit:** Read address present in cache
- **Read miss:** Read address not present in cache
- **Write hit:** Write to address present in cache
- **Write miss:** Write to address not present in cache

Read hits and misses are the simplest, and they are handled either by returning the requested data to the CPU or by first fetching the data from memory and storing it in the cache. Write hits and misses are on the other hand more difficult to handle, as they change the data instead of copying it. In addition, the goal of write handling differs from that of read handling, as the goal is to reduce bandwidth usage and not latency.

As write hits change data instead of just copying it, this requires the cache to also update the next level in the memory hierarchy with the write at some point on hit. There are two main policies [22]. One is to immediately update the level in the memory hierarchy in addition to the cache itself which is called *write through*, as the write propagates through the rest of the hierarchy immediately. The other policy is *write back* and works by only updating the cache when processing the write hit. The next level of memory hierarchy will only be updated once the cache line written to is evicted either for capacity, conflict or coherency reasons. A line written to is known as a dirt cache line, and the cache needs to track which lines have been written to. If there are several writes to the same word *Write back* reduces the updates sent to the next level compared with *Write through*. As such write- back caches reduce bandwidth for associative caches, which must always check if an address is present in a cache before updating [22].

There are also several ways of handling write misses [22]. A cache may be either *write allocate* or *no write allocate*. A *write allocate* cache will allocate a place in the cache for the line written to, while a *no write allocate* cache will simply forward the write to the next level in the memory hierarchy. In the case of write allocate caches, they must also choose between being *fetch-on-write* or not. A *fetch-on-write* cache will fetch the cache line being written to, update the cache and then perform the write operation on the cache line. In the other case, a line will be allocated but the data for that line will not be fetched. The write operation will be performed and the words in the cache line not written to will be marked as invalid. The different policies for write handling affect the miss rates of caches as well as bandwidth usage because subsequent read accesses to a line in a write-allocate and fetch-on-write cache will be hits, yet this requires additional bandwidth usage and may evict other useful data. If the program will not read cache lines written to, write-allocate and fetch-on-write will not provide benefits and will instead remove potentially useful cache lines.

The BOOM is write-back, which complicates miss handling as dirty cache lines must first be written back. However, as the BOOM is not direct-mapped, write-back results in lower bandwidth usage than write-through. For write misses, the BOOM uses a combination of write-allocate and fetch-on-write, meaning that the only difference between read and write misses is that the data must be updated after a write miss. Otherwise, they are handled the same way.

As the latency of caches changes with capacity [21], multiple levels of caches are often used to offer both low latency access for a small amount of data *and* higher latency access to a larger amount of data. Different caches in the memory hierarchy may use different policies for write handling and have different degrees of associativity.

2.2.4 Non blocking caches

As memory-level parallelism has become more important, caches have to be adapted to handle multiple requests at once. This requires a cache to be able to receive requests from the core while it is handling a miss, i. e. that a miss does not lock up or block the cache [23]. This is achieved with a Miss Handling Architecture(MHA), which contains logic for storing the missing cache line once it arrives from the next level in the memory hierarchy, as well as updating the cache. The simplest form of non-blocking cache is able to process hits while resolving a single miss and is as known as a *hit under miss* cache. Non-blocking caches are able to make use of the dual inputs of the cache by being able to send data to the CPU while receiving data from memory.

However, an MHA can be capable of resolving a number of misses without locking up. The structure in the MHA storing the information for a single miss is called a Miss Status Handling Register(MSHR). The number of misses a cache is able to handle without locking up is determined by both the number of MSHRs and their organisation. When discussing MSHR structure, the trade-off is often between the number for *primary misses* and *secondary misses* that can be supported. *Primary misses* are the first miss to a cache line, while *secondary misses* are subsequent misses to a cache line being fetched. Farkas [24] described a number of different ways to implement MSHRs, with both the *explicit* and *implicit* addressing of targets. Implicitly addressed MSHR hold a field for each word in the cache line being fetched, meaning that it can support one miss to each word in a cost-effective manner, thereby efficiently supporting as many secondary misses as there are words in a line. Explicitly addressed MSHRs are able to hold the information for a configurable number for misses to the cache line. This enables them to support multiple misses to the same word and thereby as many secondary misses as is desirable, albeit at the cost of storing more data about each miss.

An example of a Miss Handling Architecture with MSHRs is shown in figure 2.5. Upon a miss, all the MSHRs are checked to see if they are already handling a miss to that cache line. If this is the case the MSHR in question will record the

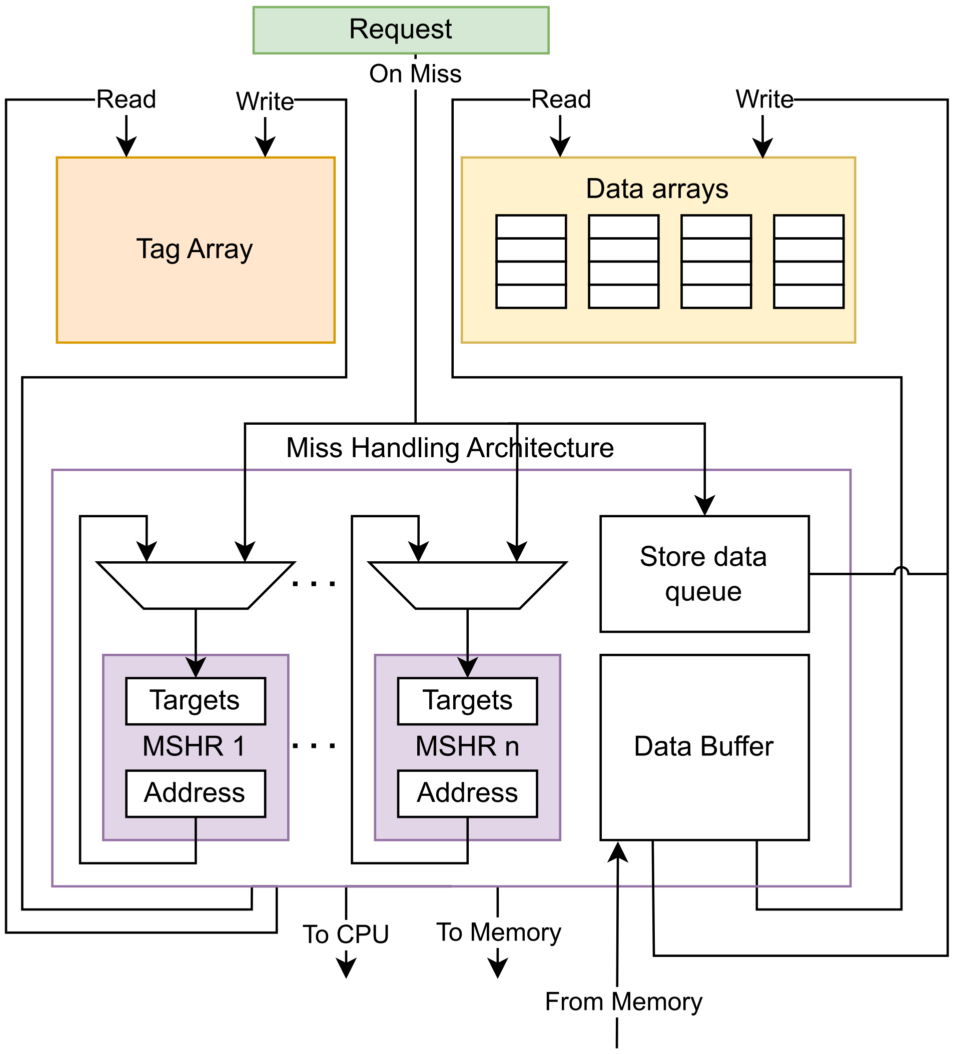


Figure 2.5: Overview of the different components of a miss handling architecture, and how they are connected. Figure taken from my Project Thesis [1].

information about the miss in its list of targets, provided there is space in its list of targets. A target typically contains information needed to forward the data to the CPU once the miss is resolved. In case no miss for that cache line is being handled, an idle MSHR is selected. If no MSHR is available the cache will lock up until a miss is resolved. Once a miss is resolved the data will be stored in a data buffer, the cache will be updated and the MSHR will forward the data to all its targets.

For write allocate caches, the Miss Handling Architecture must be extended to store the written data [19]. This data must be stored in a buffer, and read and write misses must be stored in order within the MSHRs, as reads issued before a write to the same word must not receive the data written, while reads received after must. Therefore the targets must be stored in the order they are received, at least for memory operations on the same words. The BOOM uses explicitly addressed targets and each MSHR supports a configurable number of secondary misses. The targets are stored in a queue in the order they arrived in, and write misses also store the entry in the store data buffer for the data to write. Each target stores cache metadata such as way and address, as well as its micro-operation containing data such as the corresponding entry in the LSU.

As MLP has become more important, scaling the number for MSHRs in the MHA has become ever more important [19], because the Miss Handling Architecture must be able to handle the increased number of in-flight memory instructions within the processor. The advent of multicore systems has also changed the requirements of an MHA [25]. This all makes a properly designed Miss Handling Architecture an important part of designing high-performance caches within modern systems.

2.2.5 Other cache optimisations

As the clock frequency of multiprocessors has increased, pipelined caches have been introduced to increase cache bandwidth without shrinking caches [26]. This means that caches can be accessed every cycle, even though they take multiple cycles to return the data requested. While this increases the bandwidth of the cache, the increased latency can still be mitigated. One way is using way-prediction [27]. Way-prediction seeks to use this stored data about the last way accessed, so that a cache may predict which way in a set is the next time. Using this, the cache can return the data before the tag comparison has been completed. If the way used was wrong, the data returned is invalidated and the request will be handled as usual.

Superscalar processors issue multiple instructions per cycle. By adapting caches to have multiple read and write ports, they are able to receive and respond to multiple requests per cycle. This way bandwidth can be improved and the processor can issue multiple memory instructions per cycle [28], increasing ILP. There are several ways of supporting multiple reads and writes. The most straightforward is to have multiple copies of the cache with each copy containing the same data. However, this approach has a high resource consumption and only supports a single write per cycle, as all copies must be written to. Another approach for supporting multiple cache accesses is to multi-bank the cache, where the cache is divided into several smaller banks. Each cache line is located in a single bank, and each bank has a single read-and-write port. This approach supports multiple writes as well as reads, but only for memory accesses which go to different banks. A multi-banked non-blocking cache can support multiple memory accesses from the CPU each cycle as long as there are no banking conflicts and there are available MSHRs in the Miss Handling Architecture. The MSHRs may be entirely shared between cache banks, or each bank can have its own MSHRs [19, 28].

As mentioned, cache misses can be divided into *conflict-* , *capacity-* and *cold misses*. Conflict- and capacity misses can be mitigated as described earlier. *Cold misses*, i. e. misses due to a location being accessed for the first time cannot be avoided even in a cache with infinite capacity. One scheme to avoid cold misses is to fetch the data before it is accessed. This type of prefetching can be done either in software using specific instructions or in hardware with algorithms that predict the next accessed address [29].

Chapter 3

Berkeley out-of-order Machine

3.1 Overview

The Berkeley Out-of-Order Machine(BOOM) is a superscalar out-of-order core generator, that has been progressively developed and improved over the last years [8, 30, 31]. As an out-of-order core generator, the BOOM is able to generate out-of-order cores with different capabilities based on a set of parameters, such as issue- and decode width, entries in the ROB and cache size. Therefore, BOOM cores with different parameters will have highly varying performance. Figure 3.1 shows an overview of the BOOM with indications on what parts can be configured with parameters.

As the BOOM is inspired by the Alpha 21264 [16] [8], the components are quite similar. Instructions are first fetched buffered then decoded by the Frontend. Micro-operations are then renamed and dispatched to the issue queues. The BOOM has a separate issue queue for memory, floating and integer instructions. Memory instructions proceed from the load and store queues into the L1 cache. In the case of a miss, they will then proceed to the L2 cache and in the case of another miss, they will proceed to either the L3 cache if it is configured to have one, or to the main memory. However, all memory requests must pass through the L1 cache.

3.2 Load store unit

The Load/Store unit of the BOOM is the interface to the Data Cache and is responsible for completing memory instructions after the addresses and data have been calculated. Entries in the load and store queue are allocated when the instructions are decoded. However, the addresses and data for stores are only added once it has been calculated by the store data and address generation units, as shown in figure 3.1.

Whenever the load address is ready, load instructions are sent to the data cache. At the same time, the load will compare its address with the preceding

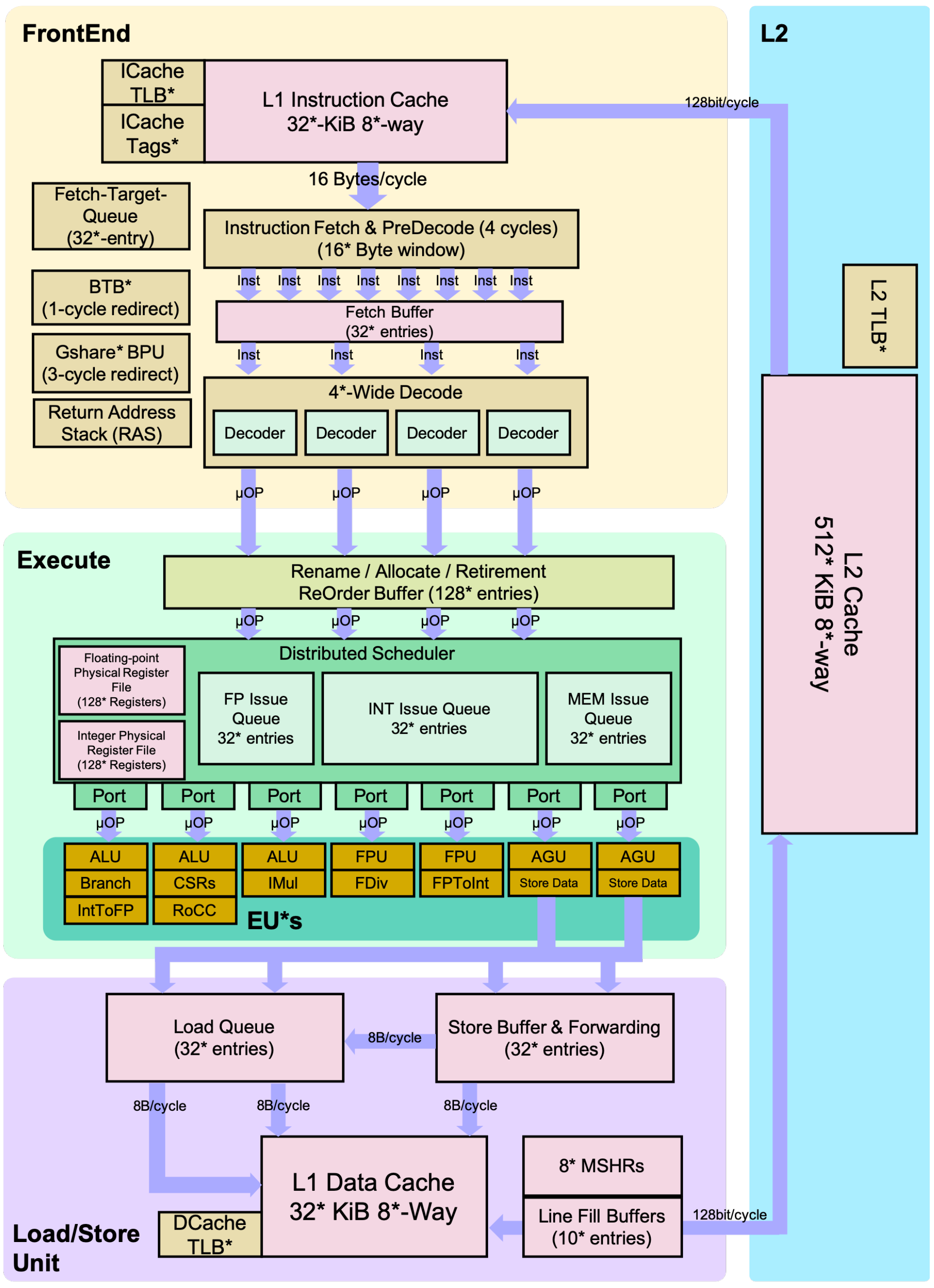

FrontEnd
ICache TLB*
ICache Tags*
L1 Instruction Cache 32*-KiB 8*-way
128bit/cycle
16 Bytes/cycle
Fetch-Target-Queue (32*-entry)
Instruction Fetch & PreDecode (4 cycles) (16* Byte window)
BTB* (1-cycle redirect)
Inst
Fetch Buffer (32* entries)
Gshare* BPU (3-cycle redirect)
Return Address Stack (RAS)
4*-Wide Decode
Decoder
µOP
L2
L2 TLB*
L2 Cache 512* KiB 8*-way
Execute
Rename / Allocate / Retirement ReOrder Buffer (128* entries)
Distributed Scheduler
Floating-point Physical Register File (128* Registers)
Integer Physical Register File (128* Registers)
FP Issue Queue 32* entries
INT Issue Queue 32* entries
MEM Issue Queue 32* entries
Port
ALU
Branch
IntToFP
CSRs
RoCC
IMul
FPU
FDiv
FPToInt
AGU
Store Data
EU*s
Load Queue (32* entries)
8B/cycle
Store Buffer & Forwarding (32* entries)
L1 Data Cache 32* KiB 8*-Way
8* MSHRs
Line Fill Buffers (10* entries)
DCache TLB*
Load/Store Unit

Figure 3.1: Overview of the components of the BOOM. Parameterisable parts are indicated with *. Taken from the SonicBoom technical report [8].

store instructions. If there is a match, the memory request will be killed and the store data will be forwarded if present in the store queue. As the data cache is pipelined, a request that is killed after one cycle will never generate nor forward data to the LSU. Store instructions are sent to the data cache only once the store itself has been committed, as data should only be written to memory once the instruction can no longer be squashed. Sending load instructions to the data cache as soon as it is ready is very important for performance, as it allows the processor to effectively hide the latency of cache misses, which is the reason the BOOM fires load requests before the address has been compared with store instructions.

Store instructions leave the store queue when they have been marked as succeeded, which is indicated by the data cache acknowledging that it has the resources needed to handle it. This is done either with an MSHR in the case of a miss or by updating the cache in case of a hit. If there are no MSHRs available, the store will immediately be resent to the data cache, and the cache pipeline will be flushed. Load instructions only leave the load queue when their data has been received. This means that loads will stay in the load queue until a miss has been resolved, while store instructions will not.

The LSU is able to send as many load instructions to the data cache as the issue width of the memory system, while only one store instruction can be sent per cycle. The L1 data cache can be configured to either offer multiple ports through having several copies of the cache, or through multi-banking. Nevertheless, the LSU only supports sending one store instruction even in the case of a multi-banked cache, contrary to how multi-banked caches are presented in [28]. The reason for prioritising loads over stores is that performance benefits significantly from having loads fired early, as previously mentioned. This is because loads often have long latencies, and other instructions depend on them. Stores on the other hand do not have other instructions depending on them, and as they are sent to the LSU only after they commit, they will only cause performance issues if the store queue fills up, causing the Frontend to stall upon decoding a store.

3.3 BOOM L1 Data Cache

The L1 data cache is the first stop for all memory requests coming from the Load/Store-unit(LSU) of the BOOM. As with the rest of the BOOM, it is highly configurable. The cache is non-blocking, as defined by[23], meaning that the cache can still handle requests from the LSU while handling a limited number of misses. The cache is write-back and fetch-on-write, as defined in [22]. This means that dirty cache lines are written back when they are evicted from the cache, and that the cache block being written to is fetched upon a write-miss. This results in data being written to the L2 only during evictions. However, both read and write requests missing will result in a cache line being fetched and a possible writeback. For applications that write data to a part of memory that it does not read from, this setup will result in the L1 cache filling up with cache lines it does not read from, and writes using needed read bandwidth due to fetch-on-write. In such a case, a policy with no-fetch-on-write would be better, yet this depends on the memory access patterns of applications.

Figure 3.2 displays the cache pipeline of the L1 cache. In the figure, the cache is displayed as having 4-ways, although the number of ways is configurable and can vary with different versions of the BOOM. As seen in figure 3.2 it is a two-stage pipelined cache, where the metadata and data are stored separately. Both the data and metadata arrays are read out in a single cycle, with tag comparison being done in the same cycle as the tag is read from the data arrays, as seen by step ❷ in figure 3.2. Way selection is performed in the second pipeline stage, along with load formatting. The cache bandwidth is defined by the memory issue width of the BOOM, with the cache being able to handle as many requests from the LSU simultaneously as the BOOM can issue memory instructions. However, only one writeback or replay may use the cache at the same time, regardless of how wide the cache is. This means that even with multiple banks, writebacks, evictions or write requests cannot access the cache at the same time. This simplifies the cache logic but may reduce performance.

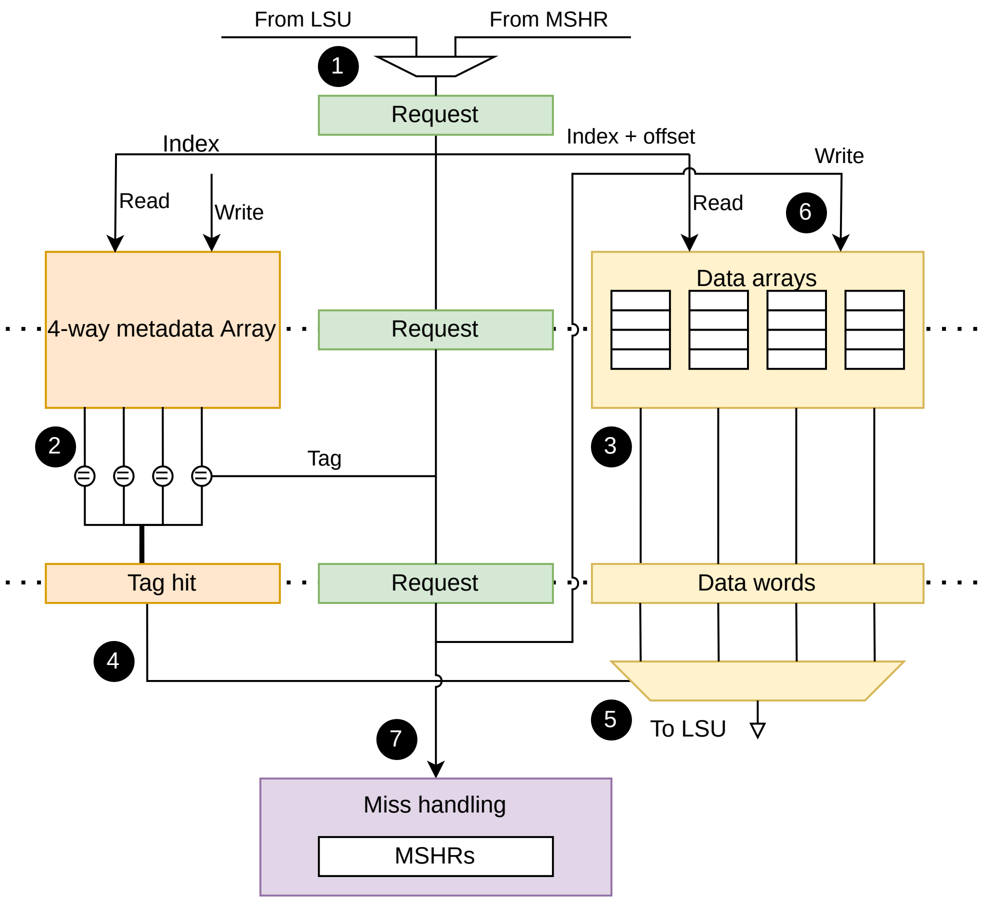


Figure 3.2: The different steps in handling a hit to the L1 data cache of the BOOM. The figure is taken from Project Thesis [1].

3.4 Hit handling

As seen in figure 3.2 a request from the LSU arrives at the cache and is used to access the cache arrays in a single cycle ❶. The data from both the metadata array and the data arrays are available in the next cycle. The tag from the request address is compared with all the tags from all ways of the metadata array, and these comparisons are forwarded to the next pipeline stage ❷. The data words read out from the data arrays are forwarded directly to the next pipeline stage ❸. In this stage the result of the tag comparisons is checked to decide if the request is a hit or a miss, as well as which way is the correct one for hits ❹. In the case of a read hit, the correct data word is selected, formatted according to the read request and sent to the LSU ❺. In the case of a write hit, the data will be written to the data arrays in the next cycle ❻. This can be done in parallel with handling a request, as the data arrays support simultaneous reads and writes. If the request is either a write- or read miss it will be handled by the Miss Handling ❼ as long as a MSHR is available. If no MSHR is available, the cache pipeline will be flushed and the LSU will be notified, yet the cache does not lock up. However, if the request is a write miss the entire cache pipeline will be flushed.

3.5 Miss handling

The Miss Handling architecture of the BOOM's L1 cache consists of a configurable number of MSHRs, a data buffer and a store data queue. Each MSHR is responsible for handling all requests to a single cache line, with each target referring to one request from the LSU. A data buffer is used for storing the cache lines temporarily between them being fetched from the L2 and stored in the data arrays. The store data queue is used for the data in store requests, i. e. the data to be written to the data arrays once the cache line has been fetched. When the cache pipeline described in the previous section detects a miss, the address of all MSHRs is compared with the address of the miss. If an MSHR is handling the same cache line and has space in its queue of targets, it will handle that miss as well. Otherwise, a free MSHR will handle the miss. While the BOOM's data cache can handle a configurable number of read requests each cycle, the MHA is only able to process one miss each cycle. This means that if a write request and a read request are processed in the same cycle both generate a miss only one of the requests will have an MSHR allocated. This does simplify the logic of the MHA, yet as noted by Tuck et. al. MHAs with low bandwidth may be a problem for extracting MLP in very aggressive Out-of-Order cores [19]. In addition, the BOOM's MHA has the restriction that it cannot have to misses with the same index at the same time.

The process of an MSHR resolving a miss is shown in figure 3.3. First, the cache line in question will be fetched from the next level in the memory hierarchy, in this case, the L2 cache ❶ and stored in the Line Buffer. The MSHR will then go through its list of targets, called the replay queue, and until the queue is either empty or it hits a store, it will send the requested data to the LSU so the data can

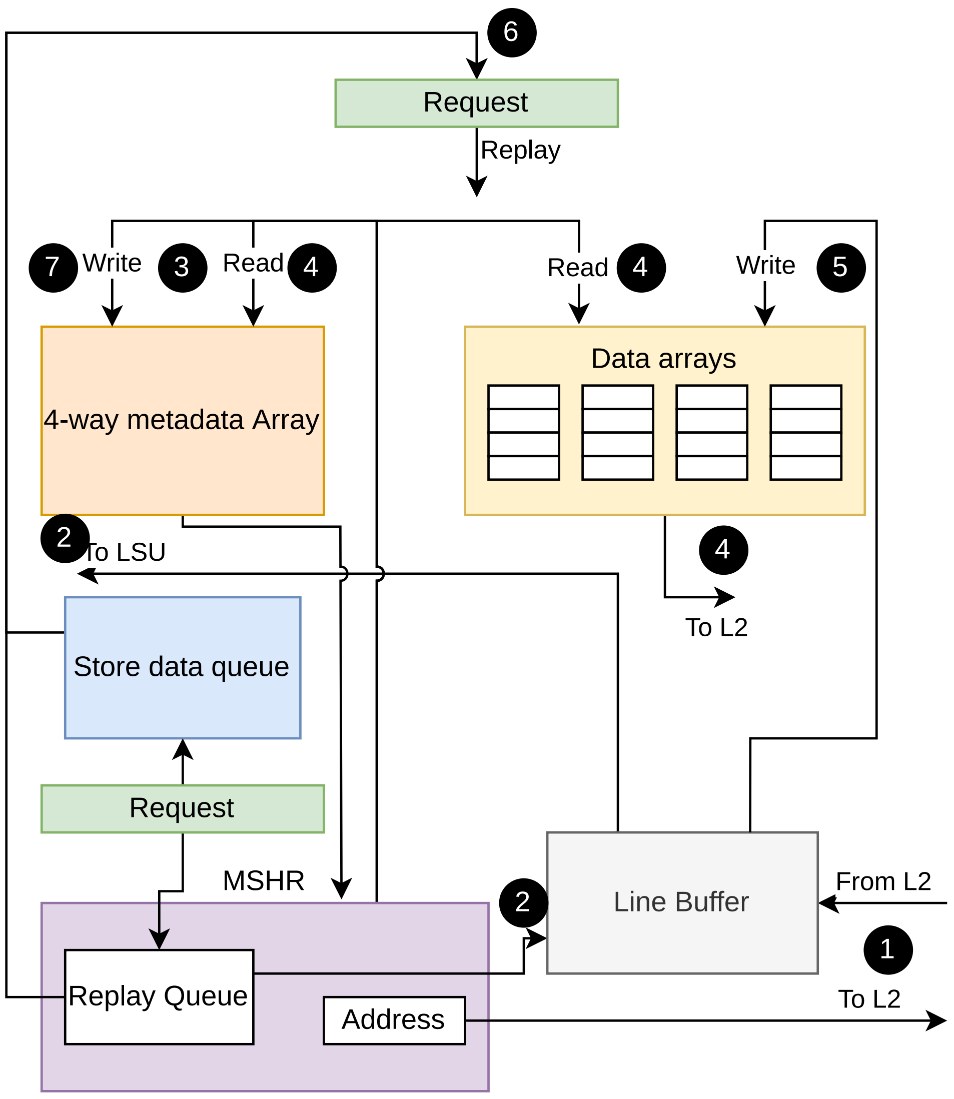


Figure 3.3: A step-by-step overview of how misses are handled in the BOOM's L1 data cache. Figure referenced from Project Thesis [1].

be used by the core ❷. This reduces the observed latency of loads significantly, as writeback may take many cycles if there is congestion at the writeback unit. The MSHR will then read out the metadata of the cache line, to check if it is clean or dirty ❸. If the cache line to be replaced is dirty, the MSHR will clear the metadata of the cache line, and get the writeback unit to write the cache line back to the L2 ❹. After the writeback is done, the MSHR will write the new cache line to the data arrays ❺. If the cache line is clean the MSHR will skip step ❹ entirely and immediately proceed to step ❺. After the data arrays have been updated, all requests still left in the replay queue, i.e. those not handled in step ❷, will be replayed as if they were sent by the LSU ❻. Once that is done the MSHR will update the metadata of the cache line just fetched ❼. The MSHR is then ready to handle another miss.

Only forwarding the data for the loads that are in front of a store in the replay queue is not strictly necessary, as a load after a store to a different word could have its data forwarded, but it is simpler. However, a load being stuck until writeback is finished could add significant latency in edge cases, possibly causing a stall if the load reaches the head of the ROB. The line buffer is used to temporarily store the data before it is written to the cache so that the cache arrays can be used for writeback or other operations while data is being received from the L2 cache. The replay queue has a configurable length, meaning that the MSHRs of the BOOM can be tweaked to support a variable number for secondary misses. The default length is 16, which is twice as many misses as there are words in a default cache line. Yet as the data cache also supports storing write misses in the replay queue, having more misses than there are words in a line is not implausible. Both Kroft [23] and Tuck et. al [19] made use of forwarding within MSHRs, meaning that a read miss could be forwarded to LSU immediately if an MSHR was already handling a write miss to that address. The BOOM instead performs this optimisation within the LSU. Forwarding in the LSU has the benefit of the data being available earlier.

3.6 Writeback handling

As the BOOM's L1 cache is a write-back cache, dirty cache blocks may have to be written back to the next level of the memory hierarchy. In the BOOM this is handled by the writeback unit. Its operation is shown in figure 3.4. First, a request for a writeback comes either from an MSHR or from the Prober ❶. If a MSHR causes a writeback this is because of a capacity conflict, while the Prober will cause a writeback when cache coherency requires it. The next step ❷ is to read out the cache line and put it into the data buffer. This is accomplished using the regular pipeline shown in figure 3.2. Once the cache line is in the data buffer, the writeback unit will notify the LSU of the writeback operation, so loads to the cache line being written back can be marked as dangerous ❸. Once the LSU has been notified, the writeback unit can write the cache line stored in the data buffer back to the next level in the cache hierarchy ❹. Depending on whether an MSHR or the Prober initiated the writeback, the data is sent as part of a Release or a

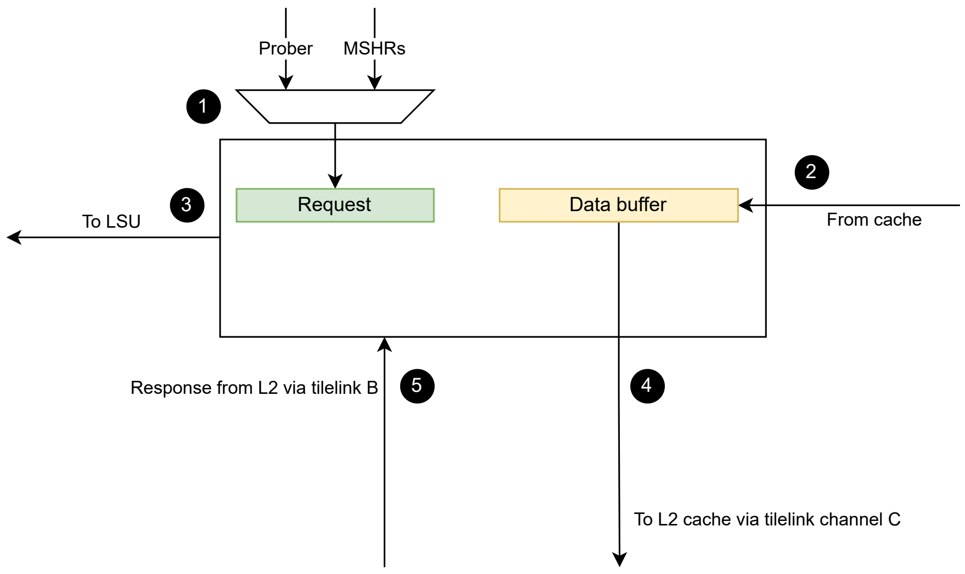

Prober
MSHRs
1
2
3
To LSU
Request
Data buffer
From cache
Response from L2 via tilelink B
5
4
To L2 cache via tilelink channel C

Figure 3.4: Overview of the BOOM's writeback unit, showing the different steps in writing a dirty cache line to the next level in the memory hierarchy.

ProbeAckData transfer operation. If the operation was a ProbeAckData this was the last step. However, in the case of a Release, the writeback unit must wait for a ReleaseAck response ❺ as per the Tilelink Protocol.

3.7 L2 Cache and DRAM

The BOOM uses the Sifive inclusive cache as the L2 cache. This cache can be generated with very different configurations in the same way as the BOOM. It handles cache coherence and is connected to other caches over Tilelink [32]. It is a highly pipelined cache that supports multi-banking with separate MSHRs per bank. It is an inclusive cache, meaning that all data contained in the L1 caches are also contained within the L2.

To complete the memory system, the Firesim simulator uses FASED to simulate the main memory [33]. FASED is a way of simulating DRAM memory in detail on an FPGA and can be configured to use different Memory Access Scheduling policies, such as First-Come-First-Served and First-Ready-First-Come-First-Served. As the memory controllers in FASED are designed in RTL they can be simulated on an FPGA the same way as FireSim, and thus offers both performance and accuracy.

Chapter 4

Tilelink

Tilelink [32] is the standard for the chip interconnect used to connect the different components in the Rocket Chip ecosystem, and therefore by extension the BOOM. Tilelink is flexible, supporting several different conformance levels, depending on what features the connected devices need. The three conformance levels in increasing complexity are:

1. TileLink Uncached Lightweight (TL-UL)
2. TileLink Uncached Heavyweight (TL-UH)
3. TileLink Cached (TL-C)

TL-UL only supports simple read and write accesses. TL-UH supports some additional features, such as atomic accesses, but still does not support caching of data. TL-C is the full version of Tilelink, with all its features, and it is used to connect the BOOM core to the next level in the memory hierarchy. It will therefore be the focus of this section.

4.1 Tilelink Cached

As mentioned, Tilelink Cached is the full protocol, and so supports all features used in the other two conformance levels [32]. It supports simple read-and-write operations, multi-cycle messages and cache block operations. The cache block operations will, unsurprisingly, be the most used operations between the L1 and the L2 caches of the BOOM. The cache block operations consist of three operations:

1. **Acquire**: Create a local copy or expand permissions of a cache line
2. **Probe**: Involuntary removes copy or permissions on a cache line.
3. **Release**: Voluntary removes copy or permissions on a cache line.

Both **Probe** and **Release** can result in the writeback of dirty data, but in the case of a **release** the writeback is voluntary, often due to capacity constraints, while in the case of a **Probe** the writeback is caused by another unit, often a higher level cache. The operations map to different actions taken by a cache, with Acquire being used to fetch a new line from memory or a higher level cache, Probe

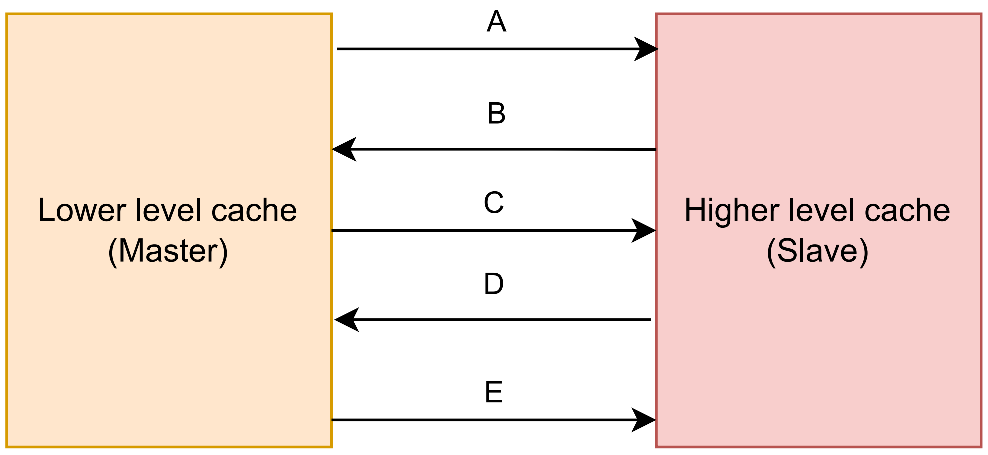


Figure 4.1: Overview of Tilelink channels along with their direction.

being used by a cache coherency scheme [34] to invalidate a local cache line and Release to remove a cache line due to a capacity conflict. As the only operation to voluntarily write data is Release, Tilelink Cached seems tailored towards write-back caches. However, TL-C also contains operations from TL-UL and TL-UH such as **PutFullData** and **PutFullData**, which simply write data to a memory address. These operations can be used by write-through caches, yet as they are not used in the BOOM they will not be discussed further.

Tilelink cached has 5 channels used in the operations mentioned above. They are channels A, B, C, D and E. Each is shown, along with its direction, in figure 4.1. Each module implementing a Tilelink interface is called an *agent*, and in a single Tilelink connection, one module is a *slave* and another a *master*. As displayed in figure 4.1, the direction of each of the channels depends on whether or not the module is a slave or a master. When it comes to caches, the master is the module that initiates Acquire operations, and when connecting two caches, the lower level cache will be the master and the higher level cache the slave, as shown by figure 4.1. Having multiple channels makes it possible to send several messages at once, meaning that a line can be written back on channel C while another is received on channel B.

4.2 Messages

Each of the operations listed above consists of a number of messages, and each message is sent over a specific channel. The messages, their response messages and their channels are shown in table 4.1. Several of the messages have separate messages for whether the message seeks to move a copy of data, or just change permissions on data already in place. All messages have acknowledgement messages that are sent once the operation is finished. To show how these messages are used, a few examples are shown below. The inclusion of acknowledgement

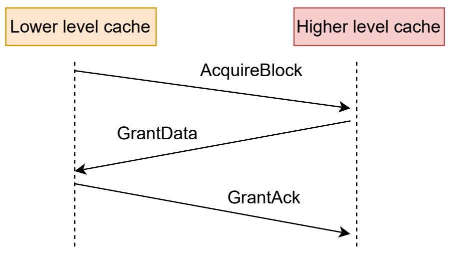
Table 4.1: Tilelink Messages, reproduced from the Tilelink specification [32].

Message	Operation	Channel	Response
AcquireBlock	Acquire	A	Grant, Grantdata
AcquirePerm	Acquire	A	Grant
Grant	Acquire	D	GrantAck
GrantData	Acquire	D	GrantAck
GrantAck	Acquire	E	-
ProbeBlock	Probe	B	ProbeAckData
ProbePerm	Probe	B	ProbeAck
ProbeAck	Probe	C	-
ProbeAckData	Probe	C	-
Release	Release	C	ReleaseAck
ReleaseData	Release	C	ReleaseAck
ReleaseAck	Release	D	-

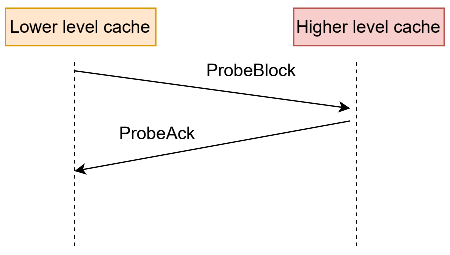
messages as the final message in each transaction is useful for avoiding deadlocks and handling faults with corrupt data, the additional messages use bandwidth and increase the duration of a transaction significantly. For example, writing back to two dirty cache lines with two releases takes longer, as an acknowledgement for the first must be received before the second writeback can begin if the node does not support concurrent Release-operations.

Subfigure 4.2a shows the message flow of an acquire. First, the Lower level cache (the master) will send an AcquireBlock message to request a copy of a block to be cached. This is typically in response to a cache miss or prefetch request to the lower-level cache. The higher-level cache (the slave) will eventually respond with a copy of the requested block. Finally, the master will respond with a GrantAck acknowledging that the transaction is finished. Subfigure 4.2b is a sequence diagram showing the flow of a probe. The slave will first send a ProbeBlock to the master requesting the removal of a specific block. This is typically done for cache coherency reasons, i. e. if another cache has written to its copy of the same block, meaning that the copy held by the lower-level cache is no longer valid. The master will then answer with a ProbeAck or a ProbeAckData depending on whether or not the block in question was dirty.

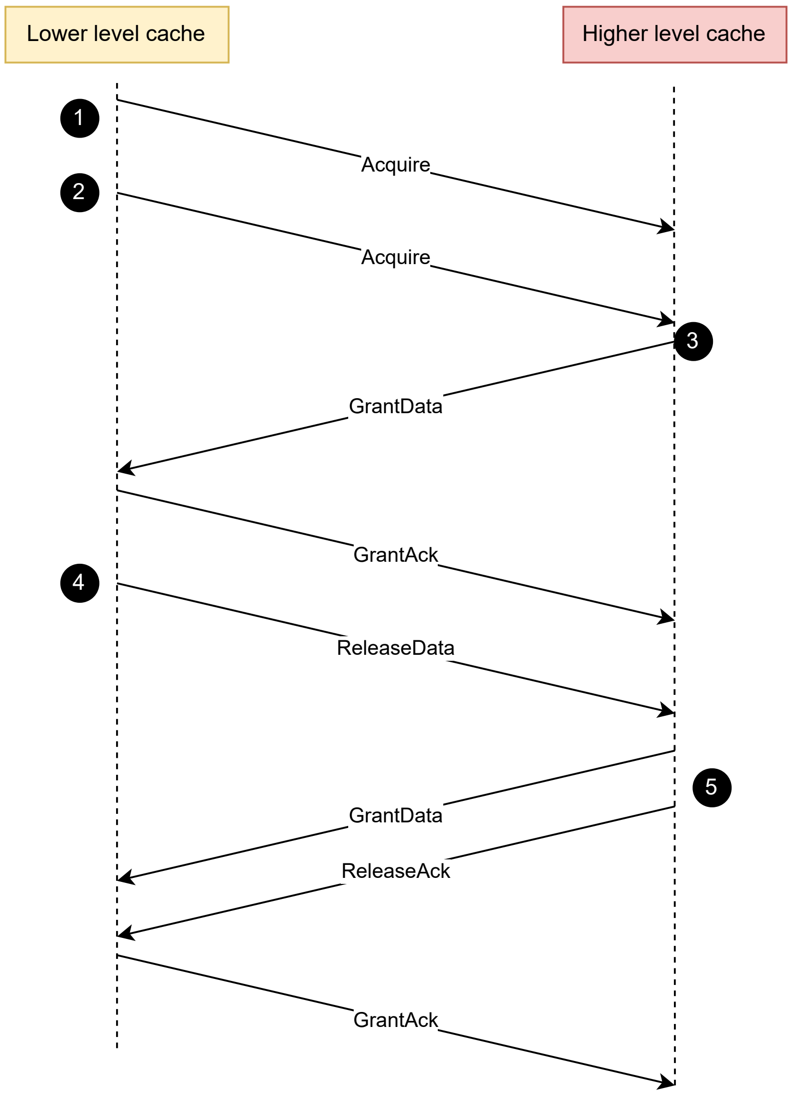
To show how a release operation works and how different operations interact concurrently a more complex example is shown in subfigure 4.2c. First ➊, the master will issue an acquire as shown in subfigure 4.2a. However, before the slave can respond the master will issue another acquire ➋. The slave will then respond to one of the acquires depending on which block it was able to read out first ➌. After the first acquire operation is completed the master will realise that it does not have space for the cache block it just received. Therefore it needs to evict a block to make space due to the conflict. To complicate matters more the block to



(a) Sequence of a acquire operation.



(b) Sequence of a probe operation.



(c) Sequence diagram of two acquires and a release operation.

Figure 4.2: Sequence diagram of Tilelink messages. The diagram on the left displays an acquire operation, the right displays a probe and the bottom displays a more complex example with multiple operations.

be evicted is dirty so it has to be written back. Because of this, the master issues a release ❹. The slave then responds to both the release and the acquire it has not yet responded to ❺. As both ReleaseAck and GrantData use channel D, they must be multiplexed. Lastly, the master responds with a GrantAck to finish the transaction.

4.3 Message components

A Tilelink message is comprised of a number of fields, each being sent in parallel over a channel. While the fields vary between each channel, they are mostly consistent for channels A, B, C and D. The fields are *opcode, param, size, source, address, mask, corrupt and data*. *Opcode* is used to identify which message is being transferred over channels. *Param* is used when transferring permissions with messages such as AcquirePerm. *Size* indicates the size of the data in the data field. *Source* is used to identify which part of the master should receive the data, i. e. the MSHR number. *Address* is simply the address of the cache block used by the transaction. *Mask* is used to mark which parts of the data field should be used. *Corrupt* indicates that the data is corrupt. The *data* field contains the data transmitted in messages such as GrantData.

The data field of a Tilelink channel is implementation specific, and the width of the data sent over may not match the width of the channel. The size of the data sent is specified by the size field. This allows high bandwidth implementation to be able to utilise large data fields, which is unnecessary for other implementations. This means that the number of beats or cycles needed to transmit a single cache line will vary depending on the width of the data channel. For example, with a data width of 8 bytes a cache line of 64 bytes will be sent in 8 beats of a burst message. However, increasing the data width to 16 bytes will reduce the number of beats to 4.

4.4 Transaction ordering

As demonstrated by the example shown in figure 4.2c, multiple Tilelink transactions may be in-flight at the same time. However, there are some restrictions on which transactions may be started while others are in flight. The Tilelink specification says the following: "All request messages generate response messages, and response messages are guaranteed to eventually make forward progress. However, under certain conditions, recursive request messages targeting the same block should not be issued until an outstanding response message is received." [32] page 68. This states that there are restrictions on request messages targeting a single block. The specification then breaks down the restriction by each operation. For Acquires and Grants, the restrictions are explicitly specified for each block. For Releases on the other hand, the protocol states: "Once the Release is issued, the master should not issue ProbeAcks, Acquires, or further Releases until it receives

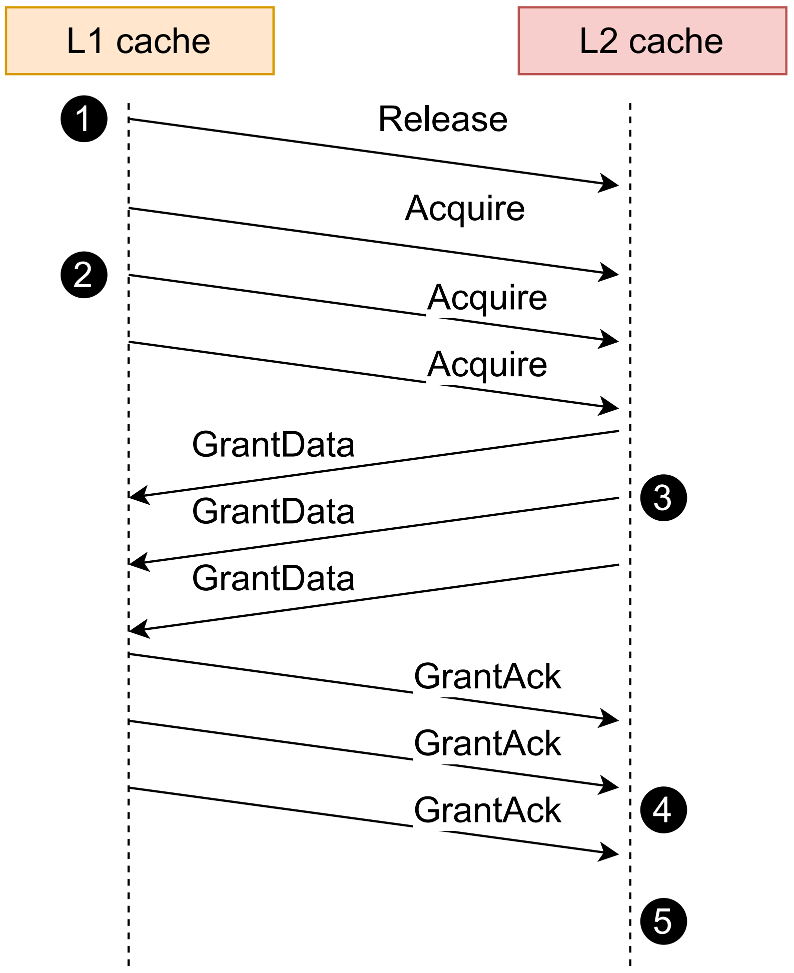


Figure 4.3: Sequence diagram describing the transaction that caused a Tilelink deadlock between the L1 and L2 caches.

a ReleaseAck from the slave acknowledging completion of the writeback." [32] page 69. While the fact that the restrictions are for operations on the same block is implied, it is not explicitly stated as it is with Probes and Acquires. The Tilelink protocol is therefore slightly ambiguous on transaction ordering for Releases,

4.5 Tilelink deadlock in the BOOM

After pipelining the BOOM's writeback unit, I encountered a Tilelink deadlock, more specifically between the L1 and L2 caches. The events that lead up to the deadlock are shown in figure 4.3. First, the L1 cache would send a ReleaseData message as it was writing back a dirty cache line ❶. After the ReleaseData was started, the L1 cache would issue several Acquires due to primary misses ❷. All Acquires would eventually be responded to with GrantData messages ❸, and the data cache would respond with GrantAcks ❹. However, in this case, the L2 cache would never respond with a ReleaseAck ❺, making the writeback unit wait until the entire core stalled. Furthermore, none of the Acquire messages targeted the same block as the Release. The deadlock would only occur after the writeback unit was pipelined and on a BOOM with 16 MSHRs. This indicates that the L2 cache would drop a Release transaction when enough Acquires were issued while it was in progress. This hints that the L2 cache interprets the Tilelink protocol to mean that no new Acquires should be issued while a Release is in progress, as one could interpret the restrictions described in the previous section. The deadlock was avoided by restricting the L1 cache to sending Acquire messages only when a Release is not in flight.

Chapter 5

Experimental setup

I use Firesim [7] in order to simulate the BOOM. Firesim is able to make cycle-accurate simulations of the BOOM and can be FPGA-accelerated in order to execute large programs in a reasonable time. However, in order to easily obtain detailed logs for the simulation of smaller program fragments, I also ran simulations of the FireSim RTL using Verilator. I used Verilator simulation to log the state of each MSHR and the writeback unit for each cycle of the simulation, which made it possible to construct the latency breakdowns shown in figure 6.3. The configuration of the BOOM core used in the simulations is shown in table 5.1. The configuration was originally developed and presented in the TEA paper [35]. The TEA configuration has a memory system that is able to exploit more MLP than the SonicBoom [8]. The rest of the core has also been upscaled to take advantage of the increased memory bandwidth.

As this thesis focuses on memory bandwidth, a way of measuring the memory bandwidth was needed. To this end, I used and configured several microbenchmarks developed by Björn Gottschall, which consisted of a main loop executing a load or store instructions to a region of memory. Examples of these loops are shown in listing 1 to 5 in section 9.1. The microbenchmarks can be configured to read or write from areas of memory with different sizes in order to stress a different level of the memory hierarchy. By running one loop which primes the memory hierarchy with the memory region the microbenchmarks use and then measuring the time it takes to either write or read an amount of data to that memory region, the memory bandwidth can be measured. Running the memory benchmarks with the same data for larger and larger memory regions allows the calculation of the memory bandwidth for the different levels of the memory hierarchy. In order to accurately measure the bandwidth, I disabled hardware prefetching when obtaining the bandwidth.

Furthermore, by configuring the microbenchmarks to only read or write to a single word per cache line, as shown in listing 2 and 4, the benchmarks stress the memory hierarchy by causing one miss per memory access. These memory benchmarks are referred to as strided benchmarks, as there is a stride between each memory access, in opposition to the streaming microbenchmarks where the addresses are adjacent in memory. The memory benchmarks can be run with load instructions, store instructions and a mix to measure the bandwidth for different circumstances. The mixed, or loadstore microbenchmark, operates on two regions of memory of equal size, reading values from the first and storing the value in the second region of memory. The load microbenchmark adds the data loaded from memory together, to avoid the loads being optimised away. The latency distributions as shown in figure 6.3 and others were obtained by logging the state of each MSHR during the microbenchmarks.

To evaluate whether the changes in bandwidth shown by the memory benchmarks could improve the performance of regular programs, I used benchmarks from the SPEC17 suite. Björn Gottschall had implemented and analysed LBM with and without software prefetching and determined that LBM experienced stalls from memory instructions in both cases. The fact that no amount of prefetching could remove stalls from memory instructions, in addition to the fact that LBM experienced stalls due to the store queue filling up [35], suggested that LBM suffered due to insufficient bandwidth. Therefore it was an excellent benchmark to evaluate changes in bandwidth.

Table 5.1: Boom configuration

Part	Configuration
Core	OoO BOOOM; RV64IMAFDCSUX @ 3.2 GHz
Frontend	8-wide fetch; 48-entry fetch buffer, 4-wide decode, 28KiB TAGE branch predictor, 60-entry fetch target queue, max 30 outstanding branches
Execute	192-entry ROB, 192 int registers, 192 float registers, 48-entry dual issue MEM queue, 80-entry quad issue INT queue, 48-entry dual issue FP queue
LSU	64-entry load/store queue
L1	32KiB 8-way I-cache, 32KiB 8-way D-cache 16 MSHRs with next line prefetcher
L2	Double banked 2 MiB 16-way L2 w/ 12 MSHRs per bank
Memory	16 GiB DDR3 FR-FCFS quad-rank, (8 GB/s maximum bandwidth, 14-14-14(CAS-RCD-RP) latencies @ 1GHz, 8 queue depth, 32 max reads/writes
OS	Buildroot, Linux 5.7.0

Chapter 6

Bandwidth analysis

6.1 Measuring Memory Bandwidth

As described in the section on caches, the latency of the memory hierarchy increases as it gets further from the CPU. Therefore bandwidth will vary depending on the memory usage of an application. By running the streaming microbenchmarks it is possible to measure the maximum bandwidth achievable read and write bandwidth to different parts of the memory hierarchy. The bandwidth is obtained by measuring the time needed to read or write 1 GiB of data to a section of memory, and then calculating the bandwidth from that by dividing GiB by the time measured. The bandwidth in GB/s is shown in figure 6.1.

To evaluate the bandwidth shown in figure 6.1, it must be compared to what the BOOM is theoretically able to achieve for different levels of the memory hierarchy. As shown in table 5.1 the BOOM used is able to issue 2 memory instructions per cycle. As the cache is pipelined, when the memory requests hit in the cache, the cache can supply two data words of 8 bytes each cycle. As the BOOM is simulated to be running on 3.2GHz, this gives a theoretical read bandwidth to the L1 data cache of $2 * 8 * 3.2 * 10^9/1000^3 = 51.1GB/s$. However, the BOOM is only able to send one write request to the data cache per cycle, even with a multi-bank configuration. As such the maximum write bandwidth to the L1 data cache is half the read bandwidth or $25.6GB/s$.

For the L2 bandwidth and beyond, the largest bottleneck must be found. All memory requests must go through the L1 data cache, and bandwidth is therefore limited to that of the L1 data cache. The L2 and L1 caches are connected via Tilelink, so the bandwidth is limited by what can be sent over Tilelink. This is configurable, yet the BOOM configuration used in this thesis has a Tilelink data width of 16 bytes. As the L2 also runs at 3.2 GHz, the L2 has the same theoretical read bandwidth as the L1 of $51.2GB/s$. However, as the L2 is larger it has a higher latency. The write bandwidth is limited by the write bandwidth to the L1, and the theoretical write bandwidth is therefore also $25.6GB/s$. The DRAM runs at 1 GHz and has the capacity to transfer 16 bytes per cycle. Therefore both the theoretical read and write bandwidth of the main memory is $16 * 10^9/1000^3 = 16GB/s$. To

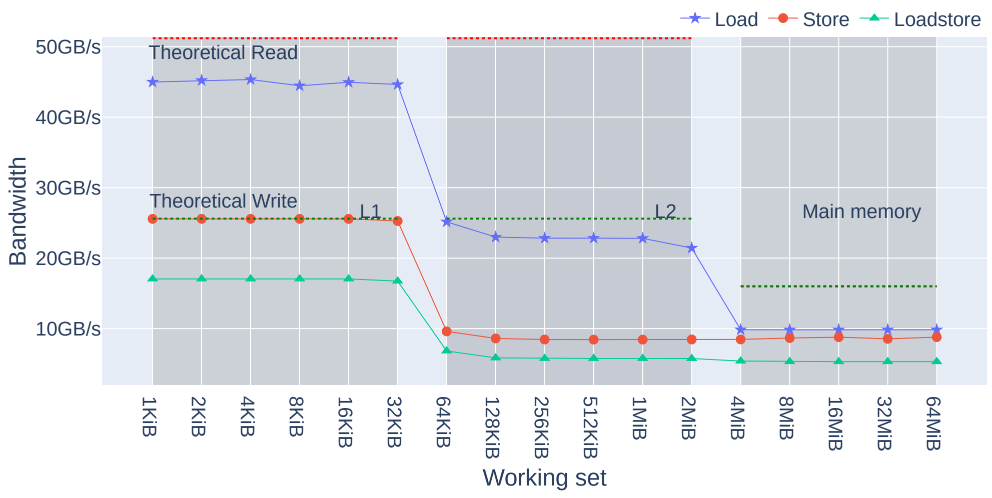


Figure 6.1: Memory bandwidth when running the streaming microbenchmark for different sizes of working sets.

summarize, the theoretical bandwidth is calculated to be the following:

- L1: 51.2 GB/s for reads and 25.6 GB/s for writes
- L2: 51.2 GB/s for reads and 25.6 GB/s for writes
- DRAM: 16 GB/s for reads and writes

Figure 6.1 shows that bandwidth varies significantly between the levels of the memory hierarchy. As the L1 cache has a capacity of 32KiB, as shown in table 5.1, this explains the drop in bandwidth between 32 and 64KiB. As the L2 has a capacity of 2MiB, this explains the additional drop in memory bandwidth between 2 and 4 MiB. When hitting in the L1 cache, the bandwidth for the load benchmark is around 45 GB/s and drops by 50% once it hits in the L2 instead of the L1. It further decreases to about 10 GB/s once it has to go to the main memory. For the store and loadstore benchmarks, the memory bandwidth decreases significantly when missing in the L1 and does not vary between the L2 and main memory.

To the L1 cache, both the read and write bandwidth measured is close to what is theoretically achievable, with both being measured over 90% of the theoretical bandwidth. However, the L2 bandwidth, and to a lesser degree the DRAM bandwidth, is measured to be far lower than what can be achieved in theory, with the load benchmark only achieving about 50% of what Tilelink is capable of transferring from the L2 cache. To explain this it is important to keep in mind that while the L2 is able to achieve high bandwidth, the latency is far larger than the L1, with the L1 having a latency of 2 cycles, while the L2 has a latency of about 14 cycles. Therefore it needs sufficient MLP to achieve high bandwidth. The number of requests it can receive is limited by the memory level parallelism supported by both the BOOM core and its data cache.

All loads stay in the load queue until the data is returned, and misses in the L1 are handled by an MSHR until resolved. Each MSHR can handle up to 16 misses

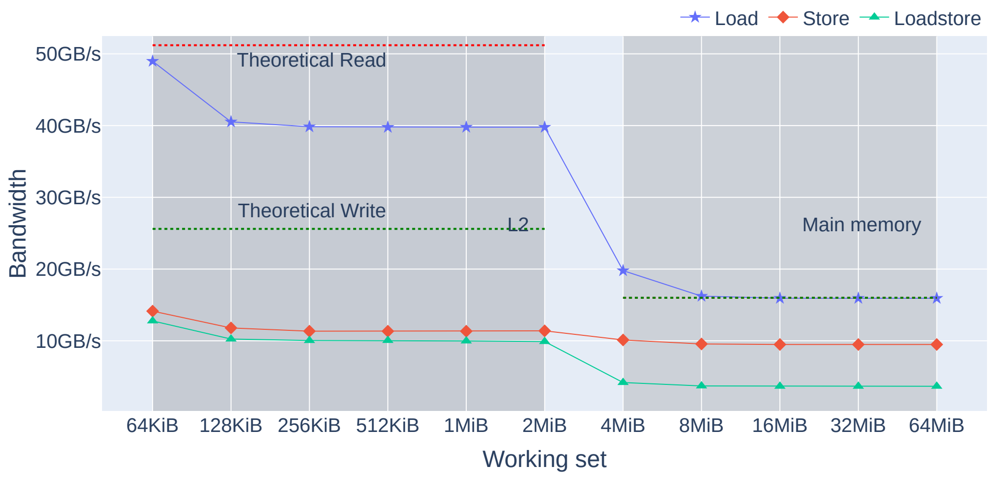


Figure 6.2: Memory bandwidth when running the strided microbenchmark for different working sets in memory.

to a single cache line while the load queue only has a capacity of 64 entries. This means that the streaming microbenchmark is only able to use 8 MSHRs at a time, as there are 8 loads to each cache line. As a result, there cannot be enough outstanding misses in the L1 to fully utilise the bandwidth between it and the L2. The write bandwidth is also limited by MLP but for a different reason. Because store instructions leave the store queue whenever they have been acknowledged by the L1 cache, its size does not affect MLP as long as it does not fill up and the Frontend can keep fetching instructions. However, only one store may be sent to the L1 at a time and the cache pipeline is often busy due to writebacks or store replays. Because of this, there is simply no room for the core to send enough store instructions to use enough MSHRs to achieve the theoretical bandwidth between the L1 and L2.

To measure the bandwidth that is achievable between the L1 and L2 the strided microbenchmark can be used, where each memory access triggers a miss, therefore generating as much MLP between the L1 and L2 as possible. As each word read or written by the core represents an entire cache line, i. e. 8 words, the bandwidth between L1 and L2 is 8 times what is measured into the core. The bandwidth between the L1 and L2 caches measured using the strided benchmark is shown in figure 6.2.

The L2 bandwidth measured with the load microbenchmark is shown in figure 6.2 and does not differ significantly from the L1 bandwidth that is shown in figure 6.1, meaning that the L1 is perfectly capable of utilising achieving read bandwidth given enough MLP. Furthermore, the read bandwidth to the main memory now reaches the theoretical maximum. For both read and write bandwidth, the first memory size used, i.e. The benchmarks with a working set of 64 KB and 4 MB perform significantly better than the rest of the memory regions for that level in the hierarchy, with the 4 MB benchmark measuring a bandwidth that exceeds the theoretical limit. This is likely due to the lower level not being sufficiently flushed, meaning that some requests hit the lower level of the memory hierarchy. Therefore, the bandwidth measured by the later regions is more representative. While the bandwidth measured by the loadstore and store microbenchmarks increases from figure 6.1, it is still far less than what is theoretically possible. The loadstore benchmark has lower bandwidth than the store one, yet that is expected as it needs to execute two memory instructions per word along with an arithmetic instruction, instead of one store instruction for the store benchmark. Nevertheless, the performance of the store and loadstore benchmarks indicate that the L1 cache has a bottleneck preventing it from achieving the theoretically possible write bandwidth.

6.2 Identifying bandwidth bottleneck

The bandwidth from the L1 cache and beyond is governed by the miss handling, as explained in section 3.5. The two factors dictating it are how many MSHRs are active at a time, and how long they spend resolving a miss, as this governs how many blocks can be fetched to the L1 per second. The time an MSHR spends resolving a miss will be referred to as the MSHR latency and is counted in the number of cycles between when the MSHR gets handed a miss to resolve and it returns to the idle state. As an example to show how the number of MSHRs and latency affect, consider how the latency changes if we double the number of MSHRs in use on average. If the latency stays the same, twice as much data will be fetched to the L1 cache, because twice as many cache lines have been fetched within the same timeframe. If the latency instead doubles, the bandwidth will stay the same, as the same number of misses will be resolved.

To discover the cause of the discrepancy between the observed and theoretical bandwidth in figure 6.2, one must therefore have to look at both the amount of MSHRs active and the MSHR latency. The bandwidth would be improved by either increasing the average amount of MSHRs active, thereby increasing MLP or by reducing the latency. By logging the state of each MSHR every cycle during one loop of the strided load and store microbenchmarks, it is possible to see how the BOOM handles memory parallelism for both eviction-heavy applications and those without evictions. From logging the MSHR states, it is possible to observe the average number of active, i.e. non-idle MSHRs and the average latency per miss. Both benchmarks have about 13 of 16 MSHRs in use on average during the benchmarks. While is this not optimal as the BOOM configuration used has 16 MSHRs, the real difference lies in the latency. The average latency per miss for the load and store microbenchmark is shown in figure 6.3.

We can see that the store benchmark suffers from far higher latency than loads, with a miss taking about 3 times as many cycles to resolve than for a load. While a slight difference should be expected because the store benchmark must evict a dirty line, as shown in figure 3.3, the addition of some extra steps does not

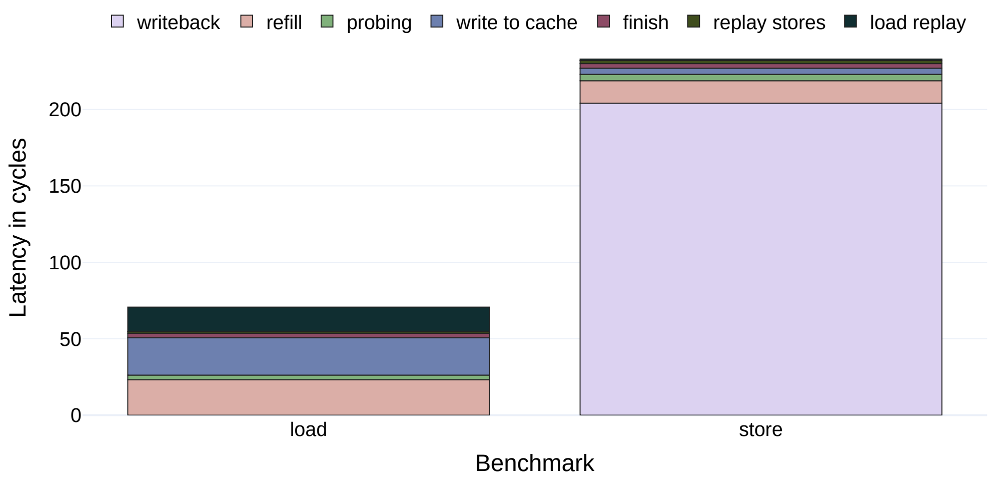


Figure 6.3: Average MSHR latency for the load and store microbenchmark broken down by different stages of resolving a miss.

explain the massive difference in latency. Clearly, the handling of misses in the store benchmarks suffers from a bottleneck not present for the misses in the load benchmarks. To understand where this bottleneck the different states shown in figure 6.3 must be discussed. The different stages shown correspond roughly to the steps shown in figure 3.3. Refill refers to fetching a cache line from the L2, load replay entails returning the fetched data to the LSU and probing is reading the cache metadata to determine if an eviction is necessary and which way to store the fetched cache line. Writeback is writing back a dirty cache line, while write to cache refers to updating the cache with the new cache line. If there were any stores to the missing cache line, they would be replayed to the cache. Lastly, finish refers to updating the metadata and clearing the MSHR state.

From the breakdown, we can see that the difference mostly comes from the MSHRs spending time on *writeback*. Only the store benchmark spends time on writeback, as it is the only one to write data to memory, thereby creating dirty cache lines which must be written back to the L2. However, an average miss spends about 200 cycles on writeback. Further examining the time each MSHR spends on writeback reveals that the MSHRs only spend 6 cycles waiting for the writeback unit to read out the cache line to be evicted before proceeding to the next step of updating the cache lines. The writeback unit can then write back the dirty cache line parallel with the cache being updated with the newly fetched cache line. The rest of the cycles spent on writeback consists of simply waiting for the writeback unit to become available. Clearly, the writeback handling is a bottleneck for eviction-heavy applications and must be improved to increase memory bandwidth for such applications. Moreover, we can break down the latency even further to show the average per miss *per MSHR*, as shown in figure 6.4. Here we can see that the latency varies wildly between MSHRs, with some MSHRs spend-

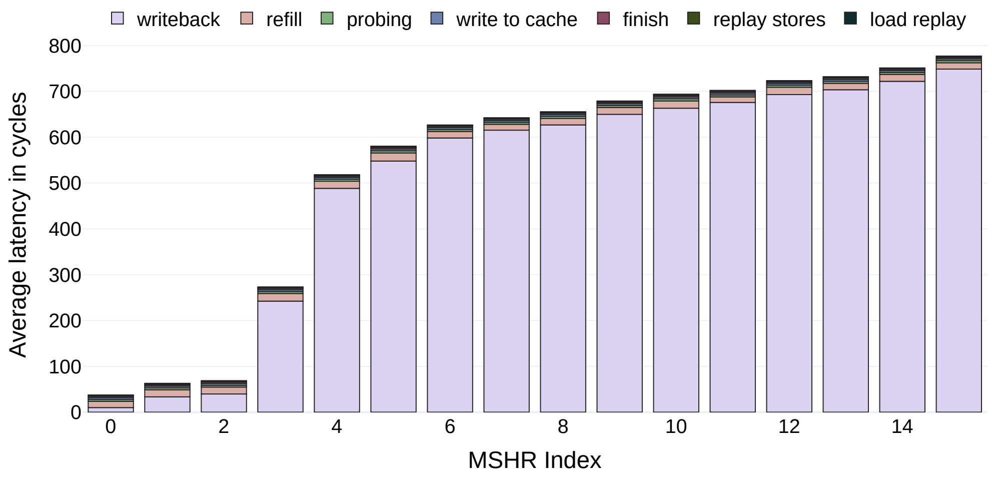


Figure 6.4: Average latency per miss for each of the MSHRs using the standard arbiter.

ing 14 times as many cycles as others resolving misses, and the difference is due to the cycles spent on writeback varying from 10 to over 700 cycles.

Chapter 7

Optimisations and evaluation

7.1 Writeback arbitration

Only a single writeback may be performed at a time, so when there are multiple units requesting to use the writeback unit at the same time, arbitration is necessary. This arbitration is handled by a standard chisel arbiter module. The arbiter in the chisel language works by having a set of multiple producers and a single consumer, where each producer has a valid output and a ready input signal. The arbiter has a single consumer which has a valid input signal and a ready output signal. Once the consumer sets ready high, it will be forwarded to one of the producers which has set valid high, and this producer's data will be sent to the consumer.

This leaves open the question of which producer should get to forward its data when multiple producers have their valid signal set to high. The standard chisel arbiter, which is used for writeback arbitration between MSHRs, has a static priority. Therefore if both MSHR number 0 and number 1 request a writeback at the same time, MSHR 0 will always get priority over MSHR 1. The second option for an arbiter in the chisel language is a round-robin arbiter. This arbiter works by giving priority in a round-robin fashion, whereby the producer with the highest priority is rotated around. If producer 0 and 1 request has valid high, producer 0 gets priority, yet the next time producer 1 would get priority. This ensures that over time, no single producer gets higher priority than the others. When there are no clear differences between the producers, this arbiter results in equal access to the consumer.

This static priority works well for cases where you always want to prioritise some signals over others, for example in the case of arbitrating between the Prober and the MSHRs for writeback, where you would want to give probes priority. However, having static priority for different MSHRs creates the uneven latency distribution seen in figure 6.4.

For very writeback-heavy applications, this can result in the time an MSHR needs to resolve a miss varying wildly depending on writeback priority. If an MSHR has a load after a store in its request queue, the load must wait for the store to be

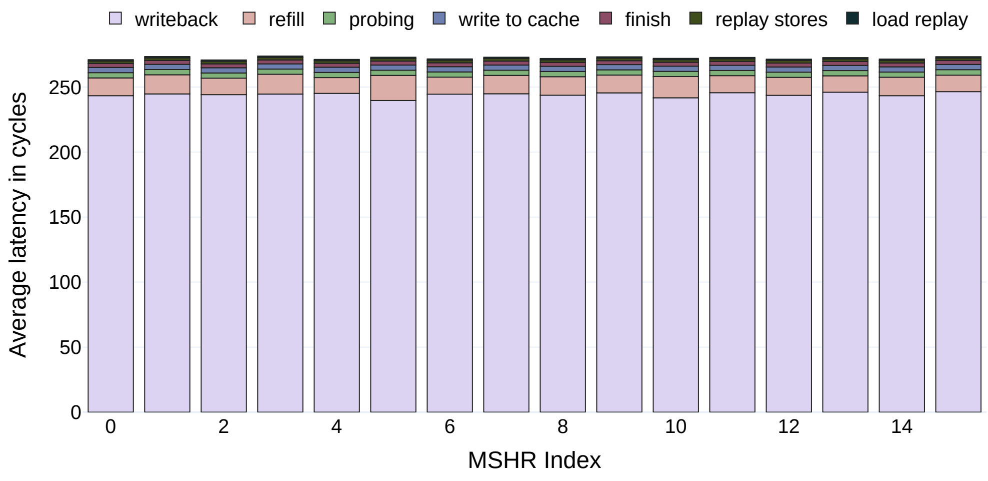


Figure 7.1: Average latency per miss for each of the MSHRs using the Round Robin Arbiter.

performed before the MSHR can resolve the load by sending data to the LSU, as described in figure 3.3. As such, a miss being handed to the "wrong" MSHR may result in a load taking significantly longer to be committed. In the worst case, this may lead to starvation as a load gets stuck at the head of the ROB and will not be committed until the new misses are not handed to the MSHRs leading to reduced MLP and stalling the entire core. Therefore, having an even latency distribution between the MSHRs is desirable. This can be achieved by using a round-robin arbiter for MSHR writeback arbitration, and the latency distribution with this arbiter is shown in figure 7.1.

The new latency is now very even. However, comparing figures 7.1 and 6.3 reveals that the average latency has increased from 232 cycles to 272 cycles. Nevertheless, bandwidth increases very slightly as the average number of MSHRs in use increases from about 13 to 15.8, meaning that the MHA is capable of using all MSHRs. Thus, replacing the arbiter has little effect on the overall bandwidth measured by any of the benchmarks. From synthesising the entire BOOM with different arbiters, as well as the two arbiters alone, the resource usage is roughly equivalent between the two. As such, the change requires few additional resources while avoiding starvation and fully utilising the MSHRs of the BOOM's MHA. To conclude, switching arbiters has little effect on memory bandwidth or resource usage, yet may avoid starvation in certain applications where a load must wait for an MSHR to complete writeback.

7.2 Reducing the writeback bottleneck

While the latency breakdown in figure 7.1 is far more even, *writeback* still dominates miss handling for an MSHR. Each MSHR spends, on average, over 200 cycles

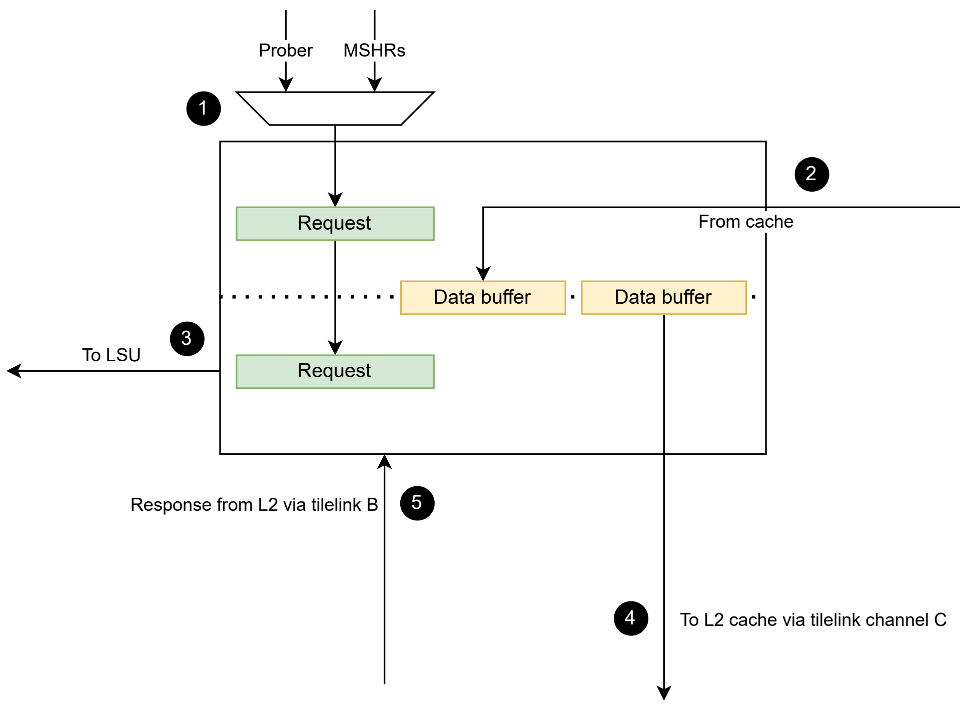


Figure 7.2: Overview of the pipelined writeback unit.

on writeback handling. As mentioned, most of this is spent waiting for the writeback unit to become available, as the MSHRs only have to wait until the writeback unit has finished reading out the cache line before continuing. This takes about 6 cycles on average, which is the minimum for this BOOM configuration described in table 5.1 because it can read out 16 bytes per cycle, and require 1 cycle for reading out of the data arrays as well as another 1 cycle for the final read to make its way through the cache pipeline. Therefore, most of the time spent on writeback is spent waiting for the writeback unit to become available. In order to reduce the time spent on writeback, the throughput of the writeback unit must be improved. As data is only being written back to the L2 cache 22% of the time, there is room to increase the throughput of the writeback unit.

7.2.1 Pipelined writeback unit

As described by figure 3.4, the writeback unit first reads out the cache line to be written back, notifies the LSU of the writeback, and then performs a Release transaction as defined by Tilelink. This entails writing the cache line to the L2 in a burst message and waiting for an acknowledgement. The writeback unit may only accept a request from another MSHR once all the steps are completed. To improve throughput, the writeback unit can be pipelined, enabling it to read out a cache line from the cache while performing a release for another cache line. The

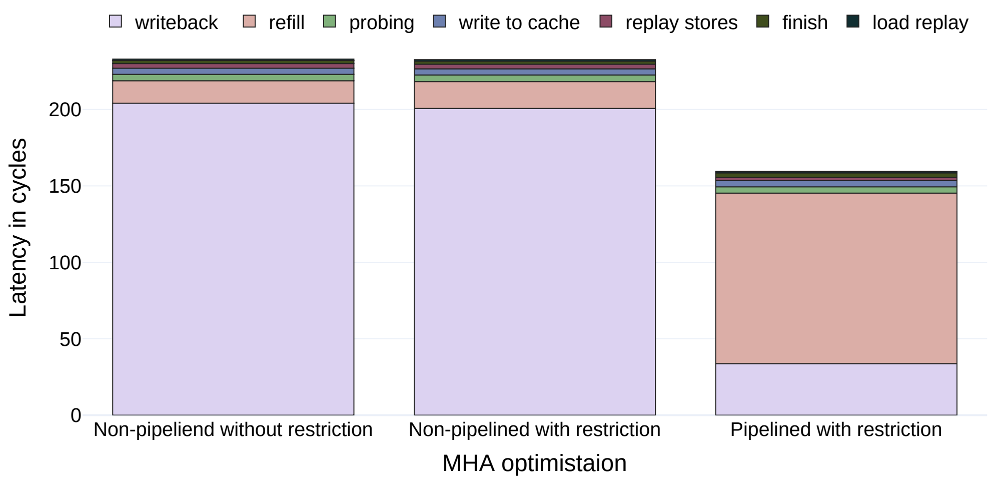


Figure 7.3: Average latency per miss for the store benchmark with a non-pipelined writeback unit with and without restrictions on issuing acquires as well a pipelined writeback unit with restrictions.

new pipelined writeback unit is shown in figure 7.2.

The pipeline is divided into two stages, where each stage handles a separate request. The first stage starts once a request is received ❶. Then the cache line is read into one of two data buffers ❷. When the next stage is idle, the request is transferred into another register, and the data buffers used by each stage are switched. The second stage first notifies the LSU of the writeback ❸, before beginning the writeback with a ReleaseData message in the case of a writeback, or a ProbeAckData message in the case of a probe ❹. If the request was a writeback, the second stage waits for an acknowledgement in the form of a ReleaseAck ❺ before returning to the idle state.

7.2.2 Effect on latency

To evaluate the effect of writeback pipelining, one must look at whether or not it reduces the time spent by MSHRs on writeback. As the introduction of the pipelined writeback unit necessitated the removal of the deadlock described in section 4.5, I must first evaluate whether the introduced restriction on issuing acquires affect latency. As shown in figure 7.3, the restrictions on acquire barely affect latency, with the exception of a slight increase in the cycles spent on refill. This is to be expected, as the MSHR must spend time waiting for a Release to be finished before issuing an Acquire. Nevertheless, the average latency remains about the same, and the bottleneck is still clearly the writeback process.

The latency with and without the pipelined writeback unit is shown in figure 7.3. The average MSHR latency is reduced from about 233 to 160 cycles a reduction of about 30%, so the writeback unit is able to significantly increase the throughput of the writeback unit. There is a stark change in the latency breakdown, with a major shift to the refill part of latency. Refill consists of issuing the acquire and waiting for the L2 cache to respond with the data. To understand the shift, I break down the refill stage into which cycles are spent before the Acquire is issued and which cycles are spent waiting on the data from L2. Both cores spend, on average, about 14 cycles waiting for the cache line from the L2. The difference lies in how many cycles are spent waiting to send the Acquire. The issuing of an Acquire itself takes only one cycle, so any additional cycles are spent waiting on being able to issue. The core with the non-pipelined writeback unit spends about 3.6 cycles, while the core with the pipelined config spends 97.7 cycles waiting. This change is explained by neither core being able to issue an acquire while a release is in progress, as explained in section 4.5. However, the pipelined write-back unit is able to have a release in progress at almost all times because it can read out cache lines in parallel, as underlined by figure 7.5. Therefore, most of the time spent in refill is spent waiting for a release transaction to complete. Because the bottleneck has shifted from which MSHR may use the writeback unit to which MSHR may issue an Acquire, this puts increased pressure on the Acquire arbitration. The situation is similar to the one described in figure 6.4, with the switch being as simple and resulting in results similar to those displayed in figure 7.1.

7.2.3 Bandwidth analysis of writeback pipelining

Figure 7.4 shows the bandwidth measured with the non-pipelined and pipelined writeback unit. As expected, the read bandwidth is not affected since the load benchmark does not need to write back any dirty cache lines, as it never writes data to memory. The bandwidth measured by the store microbenchmark is significantly increased, with an increase of 31% for the L2 and 24% for the main memory. This corresponds well with the decrease in latency observed in figure 7.3. However, the L2 bandwidth is still only about 60% of what is possible for the L2. Some overhead from writeback is to be expected, yet as shown in figure 7.3 writeback is still a bottleneck. The bandwidth to main memory is somewhat better, being about 80% of the theoretical bandwidth and is not much worse than the load bandwidth. As there will always be a little overhead due to the extra steps needed for miss handling in write-back caches, the store benchmark is expected to have a bit lower bandwidth than the load benchmark.

The loadstore benchmark only has a very slight increase in bandwidth and only to the L2. While the measured bandwidth is only slightly lower than the store benchmark without writeback pipelining, the gap increases significantly after. This indicates that the loadstore benchmark did not suffer from the same bottleneck as the store benchmark. I measured the average latency and number of MSHRs in use for a loop of the loadstore benchmark, showing that it was only able to have about 4.3 active MSHRs active at the same time, out of a total of 16. As the loadstore benchmark must execute twice as many memory instructions for the same bandwidth as the load and store benchmark, this low MLP is even more

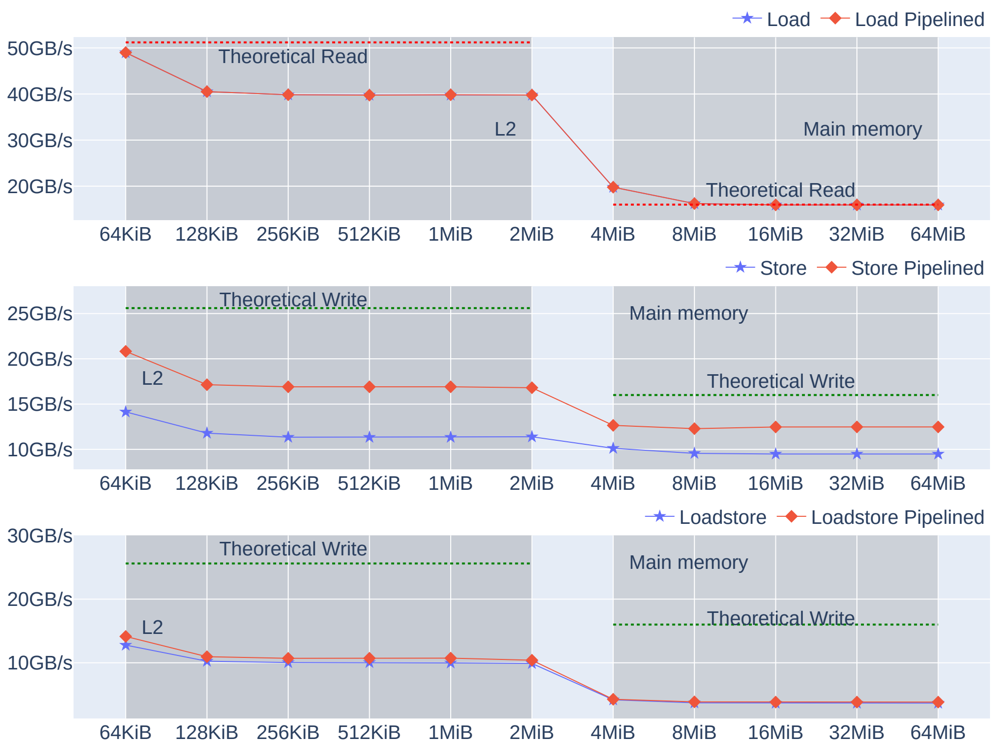


Figure 7.4: Memory bandwidth when running the different strided microbenchmarks for different sizes of memory with both a pipelined and non-pipelined wb-unit.

detrimental to performance. Further analysis showed that the BOOM's MHA interacts unfavourably with the loadstore benchmark. More specifically, the restriction on only having one outstanding miss with the same index is detrimental to the loadstore benchmark, which moves data between two regions in memory where the addresses of related loads and stores often have the same index bits. Furthermore, the fact that the MHA can only process one miss per cycle means that even though LSU can fire a load and store to memory at the same time if both miss one of them will be nacked and must be fired again later. To conclude, the bandwidth of the MHA is the major bottleneck for loadstore benchmark preventing it from exploiting MLP and achieving high bandwidth.

7.3 Increasing Tilelink width

While the pipelined writeback unit reduces overall latency by 30%, a writeback is still the bottleneck preventing the store benchmark from achieving higher bandwidth. In addition, as writebacks are continuously performed by the writeback unit, they limit MLP by preventing MSHRs from issuing acquires as soon as they are able to, as some MSHRs have to wait for about 100 cycles before being able

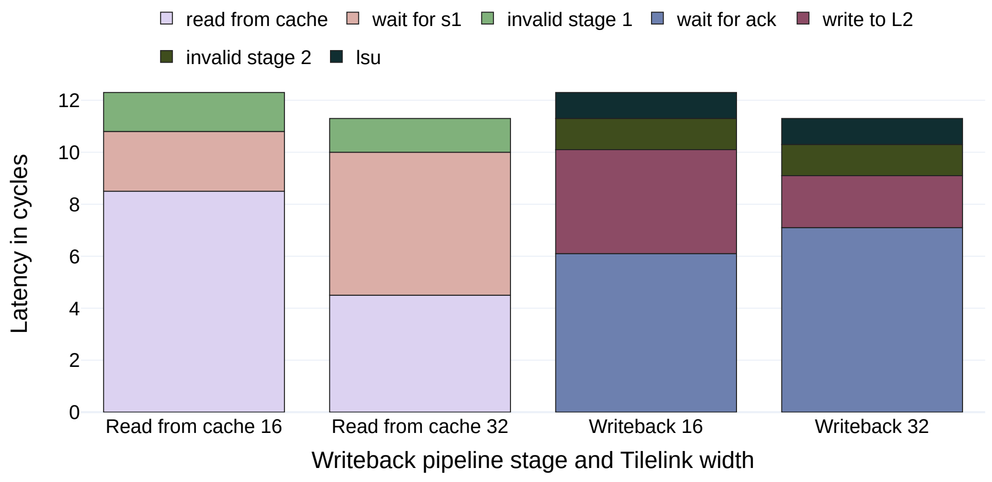


Figure 7.5: Breakdown of the time spent in different parts of the writeback process with 16 and 32 bytes per Tilelink beat. Invalid indicates that the stage of the writeback unit is idle.

to issue an Acquire. Therefore, total latency can still be reduced significantly by improving the throughput of the writeback unit. By breaking down what parts of the writeback process the writeback unit spends time on, it is possible to see which parts to optimise in order to improve throughput.

The breakdown shown in figure 7.5 shows that the writeback unit is almost always busy, as both stages of the pipeline spend only about a single cycle in the invalid state on average. Because the pipeline stages must pass through the invalid states before accepting another request this is the minimum. While it would be possible to change the writeback unit so that its pipeline stages do not need to pass through the invalid stage, it would entail additional logic and would likely only increase the time spent waiting for either the next stage or an acknowledgement. Furthermore, separating the *write to L2* and *wait for ack* would be difficult to separate, as the writeback unit must then be able to handle multiple releases in-flight at the same time, as there is no guarantee that the acknowledgements would be returned in order. This means that the writeback would need separate trackers for each Release in flight, with each Release needing a separate source ID in Tilelink, similar to how each MSHR has its own ID for Acquires. In addition, having multiple Releases in progress at the same time may lead to a deadlock similar to the one in section 4.5, as discussed in section 4.4. As the *LSU* stage takes only 1 cycle, further pipelining will not yield benefits. The time spent in *wait for ack* cannot be improved without making changes to the L2 cache. This leaves *read from cache* and *write to L2*. One way to reduce the time spent in those states is to increase the bytes written each cycle, thereby reducing the number of beats in the burst message and the number of cycles needed for a Release to finish. In addition, as the read width of the BOOMs L1 cache is determined by the width of the Tilelink data field, the time spend reading out cache lines will also be reduced.

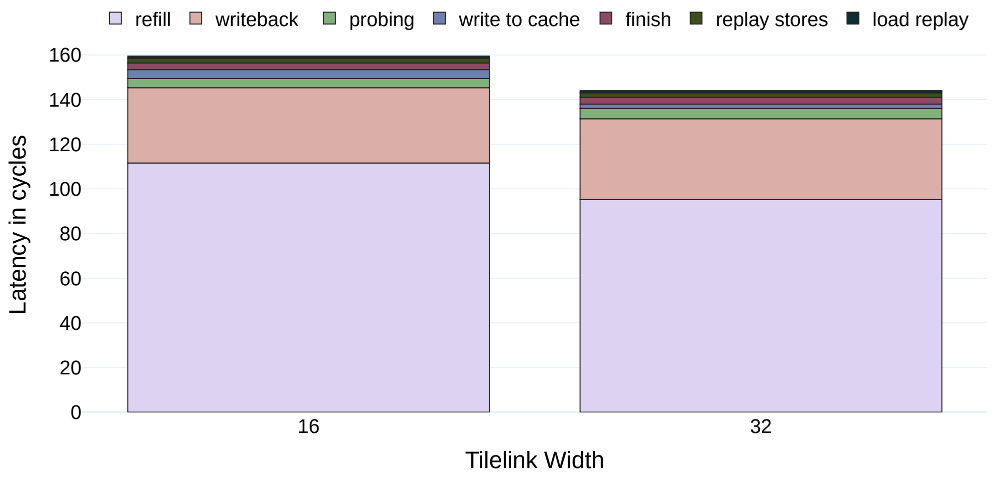


Figure 7.6: Average latency per miss for the store benchmark with 16 and 32 bytes per Tilelink beat.

The breakdown shown in figure 7.5 shows that increasing the data width in Tilelink reduces the time spent for each of the stages by about 1 cycle, and increases the portion of time spent waiting for the L2. Average latency per miss is also decreased, as shown by figure 7.6. There is now little room for improving the throughput of the writeback unit by altering it, as most of the time is spent waiting for acknowledgement from the L2. As the L2 is highly pipelined there is a minimum of 4 cycles before a ReleaseAck can be received. As the ReleaseAck and GrantData messages must be multiplexed over channel D, this often causes additional cycles spent waiting for an acknowledgement. Therefore, it is difficult to reduce the time it takes for a ReleaseAck to be received.

As expected from the slight reduction in latency shown in figure 7.6, the bandwidth measured in the store microbenchmark increases slightly as shown in figure 7.7. The bandwidth to main memory does not increase significantly, as it was already close to what was possible with the pipelined writeback unit, indicating that the maximal write bandwidth to main memory has been reached. The L2 bandwidth does increase slightly, yet is still only about 2/3s of what is achievable to the L1. As shown in figure 7.6 the writeback process still takes longer than it needs when missing in the L1 due to writeback. Therefore, bandwidth can still be improved by reducing the writeback bottleneck. One way to do this would be to separate the miss handling of multiple cache banks so that each may issue writebacks in parallel. Unfortunately, the BOOM does not currently support this, as cache banks do not alter the miss handling. This is underlined by performance not increasing as shown in figure 7.7.

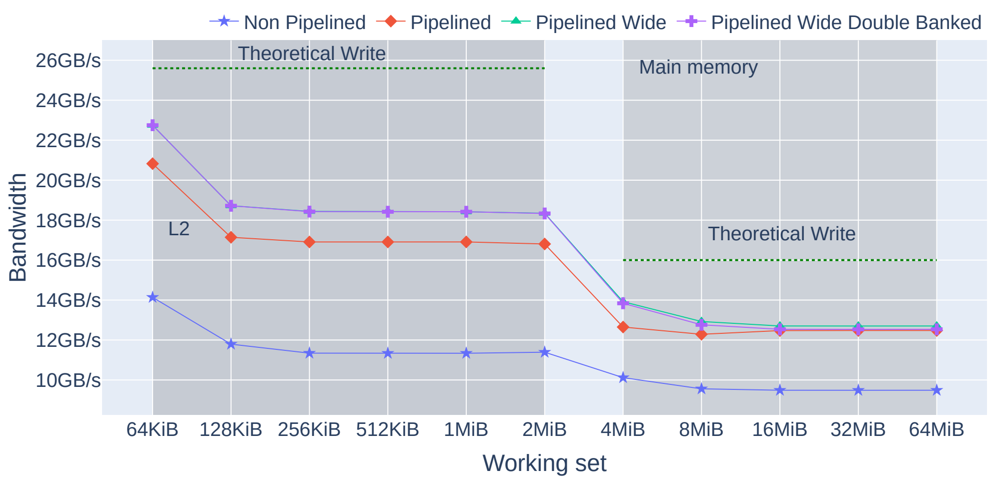


Figure 7.7: Memory bandwidth into the L1 for the store benchmarks with different memory sizes.

7.4 Resource consumption

To evaluate if the performance increase is worth it, one must look at the increase in resource usage. The resource usage of the entire synthesized BOOMs is shown in table 7.1, with the increases shown as well. The FPGA resources not shown were unchanged between the BOOM with the standard writeback unit and the BOOM with the pipelined one. As we see from table 7.1 the new writeback unit uses very few additional resources with there only being a slight increase for LUT and flip-flops. The Tilelink widening uses significantly more resources. This can be explained by the read width of the SRAM modules in the data cache being the same as the Tilelink width, as well as there being a Tilelink buffer between the L1 and L2 caches. Therefore it is not sufficient to increase the channel width, but other more expensive changes must also be made.

Table 7.1: Resource usage of the BOOM with the different configurations used.

Resource	Base	Pipelined WB	Diff	Wide Tilelink	Diff
LUT	838851	840276	0,170 %	893980	6,4 %
LUTRAM	19884	19884	0	24820	24,9%
Flip-flops	357170	357799	0,176 %	381212	6,5%
BRAM	813,50	813,50	0	408,50	-50%

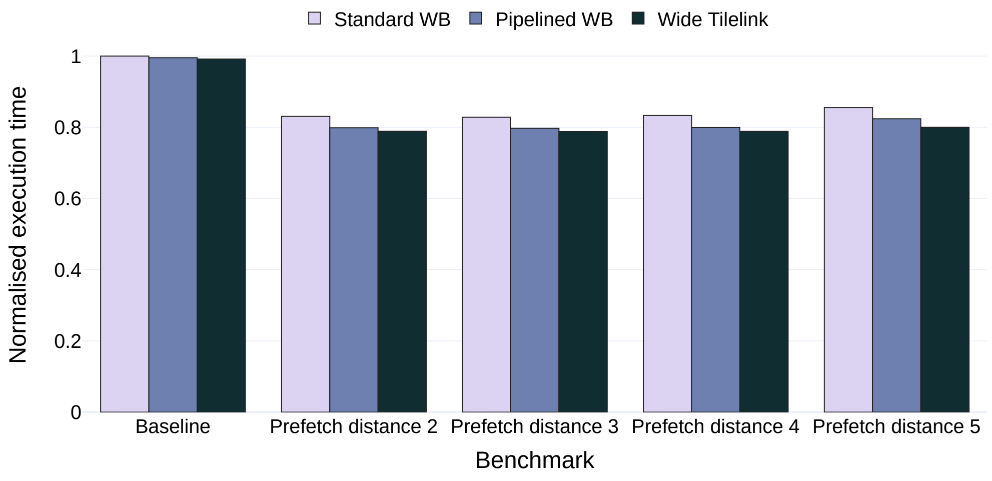


Figure 7.8: Execution time of LBM benchmarks with different software prefetching distances. Normalised to baseline with standard writeback unit.

7.5 LBM performance

To evaluate if the increase in bandwidth is useful for the performance of actual programs, I consider the performance of LBM with and without the pipelined writeback unit and the increase in Tilelink width. The execution time in cycles for each of the programs on the different BOOM-cores is shown in figure 7.8. Both the pipelined writeback unit and the widening of the Tilelink channel increase performance for all programs. However, the increase is not uniform, with the greatest increase being for the variants of LBM with software prefetching, namely *Prefetch distance 2 - 5*, where distance represents the number of loop iterations which data is prefetched for. This would indicate that LBM suffers from a bottleneck other than write bandwidth, with write bandwidth only being the bottleneck when prefetching is introduced. Nevertheless, the pipelined writeback unit and wide Tilelink still reduce execution time for the baseline by 0,47% and 0,83% respectively. The program with prefetching distance 3 has the best performance of the programs for all optimisations, and the pipelined writeback unit increases performance by about 3,8%. However, with a Tilelink-width of 32, the versions of LBM with prefetching have roughly the same execution time regardless of the prefetch distance.

The cycle stacks for the different benchmark executions are shown in figure 7.9. I obtained the cycle stacks using methods developed by Gottschall et. al. [35]. The *INT*, *FP* and *MEM* categories represent time spent on executing the different types of instructions. Hence, they naturally compose the largest portion of execution time and since LBM is a floating point-heavy benchmark, spending most of its time executing floating point instructions is to be expected. As there are many floating point instructions with true dependencies, it is difficult to avoid spending a large amount of time on floating point execution. *ST-LLC, ST-TLB and ST-L1* rep-

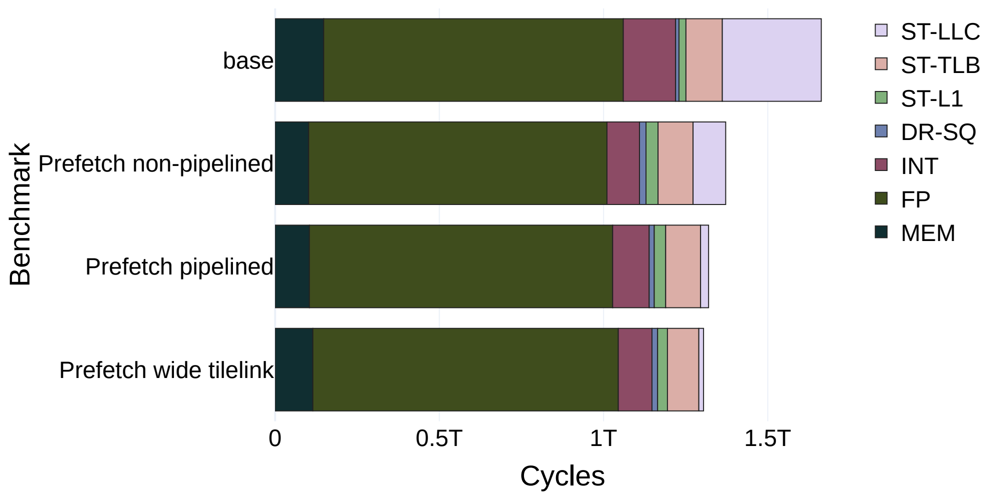


Figure 7.9: Cycle stacks of the main function for the exp3 and baseline LBM benchmarks for different optimisations. Cycle stacks for baseline do not vary much with optimisation, so only the non-pipelined results are shown.

resent stalls, i. e. cycles where no instruction is committed, caused by a miss to the LLC, TLB and L1 cache respectively. *DR-SQ* represent a drain, i. e. when the ROB empties due to the frontend locking up, caused by the Store Queue being full and thus preventing new instructions from being dispatched. While the largest number of cycles are devoted to the execution of floating point instructions, the baseline suffers from a lot of stalls due to misses in the LLC, which in this case is the L2 cache. Prefetching significantly reduces the stalls from LLC misses, yet it remains the largest source of stalls. Pipelining further reduces LLC stalls by about 75%, with the wide Tilelink almost removing them entirely. The number of cycles not spent on stalls increases slightly with writeback pipelining, as some of the execution cycles that were previously hidden behind stalls now come to the front. With writeback pipelining the largest source of stalls is the TLB. Nevertheless, there are still a significant amount of stalls and Frontend lock-ups from the memory system, such as stalls from L1 and LLC misses as well as the store queue filling up. This indicates that there is still room for improvement in the BOOM's memory system. In addition, the presence of stalls from LLC and L1 misses with prefetching indicates that performance would be improved with additional bandwidth, as could in theory completely hide the latency from LLC misses by prefetching early enough with sufficient bandwidth.

Although the cause for performance improvement in LBM *with prefetching* is clear, the cycles stacks offer no clear explanation for the lack of improvement in LBM *without prefetching*. The cycle stacks for the baseline benchmark are similar to the one shown in figure 7.9, regardless of optimisation. As the bottleneck is the memory instructions, as indicated by the number of stalls due to LLC misses, one would expect increased memory bandwidth to reduce the number of stalls and improve performance, yet this is not the case. The fact that prefetching is needed to make LBM bandwidth bound suggests that the BOOM is not able to fully utilise the available bandwidth without it. Therefore, there is likely a bottleneck preventing the BOOM from exploiting enough MLP to achieve high bandwidth.

As the Tilelink widening gives a smaller increase in performance than the pipelined writeback unit on its own, for significantly more resources, increasing it is not cost-effective. In addition, the theoretical read bandwidth between the L1 and L2 now exceeds the bandwidth between the L1 and the BOOM core. Therefore, the memory issue width of the BOOM core should be increased before Tilelink is made wider so that the read bandwidth to the core would increase as well.

7.6 Other optimisations

The main bottleneck for the store microbenchmark is still the writeback process, as shown in figure 7.6 and LBM would benefit from increased bandwidth as indicated by the memory stalls in figure 7.9. The main bottleneck is that only a single release transaction can be in progress at a time. As the writeback unit spends most of its time waiting for acknowledgement from the L2 cache, shown in figure 7.5, relaxing this constraint would make it possible to overlap releases in the same way as Acquires. One way of having multiple releases in parallel would be to alter the miss handling architecture of the data cache so that each cache bank has its own MSHRs, as described by Tuck et. al. [19]. Each bank may then be its own Tilelink agent connected to the L2, and may then issue Tilelink transactions separately. While this would improve bandwidth, imbalanced accesses and misses to the cache banks may lead to only one of the banks being used, resulting in the same bottleneck observed in this thesis. In addition, each bank would likely have fewer MSHRs than in a unified miss handling architecture, and imbalanced access could therefore cause the cache to lock up. On the other hand, the introduction of address mappings similar to those developed for DRAM [36] may reduce lock-ups due to imbalanced access.

Further increases in bandwidth may be accomplished by overlapping the writeback of dirty cache lines with fetching new cache lines, instead of doing these steps sequentially. This is specifically mentioned as a possibility in the Tilelink specification [32], as there is no restriction on issuing a release while an Acquire is in flight. Overlapping the stages will mean doing the probing and writeback stages of miss-handling while waiting for the new cache line from the L2. For workloads with few writebacks and thus low contention for using the writeback unit, this could hide the entire latency of writebacks under the fetch latency from the L2. However, it might also result in more misses, as instructions might use the evicted cache line.

Tuck et. al. also underlined the importance of having an MHA with high bandwidth in the sense that it can process many cache misses per cycle [19]. While this was accomplished with a multi-banked MHA to allow for multiple reads and writes to the MSHRs each cycle, this is not necessary for the BOOM. While the

BOOM's MHA is only capable of handling one cache miss each cycle, the address of all misses is sent to the MSHRs to check for a match. Extending the MHA to process more than one miss per cycle would mean that fewer requests from the LSU would have to be sent to the cache multiple times before being handled. Another change to MHA that may offer increased performance for certain programs is removing the restriction on not having two outstanding misses with the same indexed bits. Both of these would reduce the time it takes for MSHRs to be allocated, making it easier to exploit MLP and hide the latency of cache misses. As LBM requires prefetching to be bandwidth bound, this may increase performance as loads will be executed earlier.

Lastly, the BOOM data cache only supports a single write per cycle, even when having multiple banks. As pointed out by Sohi [28], being able to support multiple writes per cycle is an advantage of multi-banked caches as opposed to having duplicate caches which is the other option in the BOOM. Therefore, making the Load/Store unit and the data cache able to support multiple writes per cycle in a multi-banked configuration could improve performance, especially if the memory issue width were to be increased further. As LBM suffers from the store queue filling up, which causes the frontend of the core to stop dispatching instructions, draining the ROB, making it easier for stores to leave the store queue would improve performance. In addition, stores cannot be sent to the L1 cache while the writeback unit is reading out a cache line or an MSHR is replaying either loads or stores as the L1 cache is hard-wired to only accept one of these requests at a time. As such, making it possible for the L1 cache to process stores while reading out a cache line or replaying load requests would also avoid the store queue filling up.

Chapter 8

Conclusion and Further work

8.1 Conclusion

In this thesis I explain the Berkeley Out-of-order Machine's memory system in detail, the Tilelink protocol and different cache organisations and their effect on performance. Furthermore, I evaluate the bandwidth for different levels of the memory hierarchy, addressing task 1 in section 1.2. This evaluation highlights that the BOOM delivers insufficient memory bandwidth when going beyond the L1 for eviction-heavy applications. With further analysis, I conclude that this is caused by the writeback of dirty cache lines becoming a bottleneck when handling many concurrent misses that require dirty cache lines to be written back. As the writeback unit cannot concurrently read out a cache line while writing back another dirty cache line, its throughput is limited which causes the mentioned bottleneck.

To improve the memory bandwidth I develop an improved pipelined writeback unit capable of reading out the next line to be written back concurrently with writing another cache line to the L2, increasing throughput significantly. By identifying and alleviating this bottleneck I address task 2 as defined in section 1.2. This writeback unit increases the bandwidth for eviction-heavy applications by 20 to 30% for only a 0,2% increase in resource consumption, addressing task 3. Lastly, small changes are made to writeback arbitration to avoid starvation for certain applications without an increase in resource consumption. All these changes are then evaluated on LBM with and without prefetching, to show an almost 4% increase in performance for LBM with software prefetching, addressing task 4 as described in section 1.2. While evaluating the pipelined writeback unit, I encounter and fix a Tilelink deadlock in the BOOM where the L2 cache would not acknowledge a writeback if enough new cache lines were requested while the writeback was in progress.

8.2 Future work

The evaluation with LBM in section 7.5 shows that there is still room for improvement in performance, especially for the version without prefetching, which is not bandwidth bound. Based on this, I suggest several areas of the BOOM with could benefit from improvement.

8.2.1 Data cache and Miss Handling Architecture

While the number for MSHRs and the width of the BOOM's data cache is configurable, the number of misses that can be handled each cycle is not. As the number of MSHRs is increased, it becomes ever more difficult for the BOOM to use them all at a time. In addition, the bandwidth from the loadstore benchmark in figure 7.4 shows that the restriction on only being able to handle a single miss with the same index bits at the time may impact performance. Lastly, the fact that the data cache can only handle a single store per cycle, and only when it is not performing an eviction or replaying misses may lead to the store queue filling up even when MSHRs are available, as underlined by the performance impact shown in figure 7.9. All this shows that improvements to the BOOM's data cache and MHA would likely improve performance, especially for programs which exploit a large amount of MLP.

8.2.2 Write bandwidth

The write bandwidth presented in figure 7.4 is still below what is theoretically feasible and LBM with software prefetching still suffers from memory-related stalls in figure 7.9. Therefore, the write bandwidth achieved by the pipelined writeback unit is not sufficient. Writeback is still the bottleneck for achieving higher bandwidth, as shown by figures 7.3 and 7.6. As the writeback unit spends most of its time waiting for ReleaseAck messages as per Tilelink, adding support for having multiple Releases in flight concurrently would reduce the writeback bottleneck. This could be done by either having separate miss-handling for each cache bank or by enabling the writeback unit to issue a Release for a cache line while waiting for a ReleaseAck for another.

Bibliography

[1] E. F. Nesset, *Analysing the memory hierarchy of the BOOM*, English, Dec. 2022.

[2] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi and S. Reinhardt, 'The M5 Simulator: Modeling Networked Systems,' *IEEE Micro*, vol. 26, no. 4, pp. 52–60, Jul. 2006.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood, 'The gem5 simulator,' *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[4] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon and É. F. Zulian, 'The gem5 Simulator: Version 20.0+,' *arXiv preprint arXiv:2007.03152*, Sep. 2020.

[5] T. E. Carlson, W. Heirman and L. Eeckhout, 'Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation,' in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, Association for Computing Machinery, Nov. 2011, pp. 1–12.

[6] R. Panda, S. Song, J. Dean and L. K. John, 'Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?' In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2018, pp. 271–282.

[7] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach and K. Asanovic, 'FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud,' in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2018, pp. 29–42.

[8] J. Zhao, B. Korpan, A. Gonzalez and K. Asanovic, 'SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine,' *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, en. Morgan Kaufmann, Nov. 2017, ISBN: 978-0-12-811906-8.

[10] W. A. Wulf and S. A. McKee, 'Hitting the memory wall: Implications of the obvious,' en, *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[11] D. A. Patterson, 'Latency lags bandwith,' *Communications of the ACM*, vol. 47, no. 10, pp. 71–75, Oct. 2004.

[12] Yuan Chou, B. Fahs and S. Abraham, 'Microarchitecture optimizations for exploiting memory-level parallelism,' en, in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, Munchen, Germany: IEEE, 2004, pp. 76–87.

[13] W. M. Johnson, 'Super-Scalar Processor Design,' en, Stanford University, Technical report, Jun. 1989.

[14] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras and L. Eeckhout, 'The load slice core microarchitecture,' en, in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ACM, Jun. 2015, pp. 272–284.

[15] R. M. Tomasulo, 'An Efficient Algorithm for Exploiting Multiple Arithmetic Units,' *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, Jan. 1967.

[16] D. Leibholz and R. Razdan, 'The Alpha 21264: A 500 MHz out-of-order execution microprocessor,' in *Proceedings IEEE COMPCON 97. Digest of Papers*, Feb. 1997, pp. 28–36.

[17] S. Palacharla, N. P. Jouppi and J. E. Smith, 'Complexity-effective superscalar processors,' in *Proceedings of the 24th annual international symposium on Computer architecture*, ser. ISCA '97, May 1997, pp. 206–218.

[18] S. Eyerman, L. Eeckhout, T. Karkhanis and J. E. Smith, 'A mechanistic performance model for superscalar out-of-order processors,' en, *ACM Transactions on Computer Systems*, vol. 27, no. 2, pp. 1–37, May 2009.

[19] J. Tuck, L. Ceze and J. Torrellas, 'Scalable Cache Miss Handling for High Memory-Level Parallelism,' in *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Dec. 2006, pp. 409–422.

[20] A. J. Smith, 'Cache Memories,' *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, Sep. 1982.

[21] P. Shivakumar and N. P. Jouppi, 'CACTI 3.0: An Integrated Cache Timing, Power, and Area Model,' en, Western Research Labs, Compaq, Technical report, Aug. 2001.

[22] N. P. Jouppi, 'Cache write policies and performance,' *ACM SIGARCH Computer Architecture News*, vol. 21, no. 2, pp. 191–201, May 1993.

[23] D. Kroft, 'Lockup-free instruction fetch/prefetch cache organization,' in *Proceedings of the 8th annual symposium on Computer Architecture*, ser. ISCA '81, May 1981, pp. 81–87.

[24] K. I. Farkas and N. P. Jouppi, 'Complexity/performance tradeoffs with non-blocking loads,' *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2, pp. 211–222, Apr. 1994.

[25] S. Li, K. Chen, J. B. Brockman and N. P. Jouppi, 'Performance Impacts of Non-blocking Caches in Out-of-order Processors,' en, HP Labs Tech, Technical paper HPL-2011-65, 2011.

[26] A. Agarwal, K. Roy and T. Vijaykumar, 'Exploring high bandwidth pipelined cache architecture for scaled technology,' in *Automation and Test in Europe Conference and Exhibition 2003 Design*, Mar. 2003, pp. 778–783.

[27] D. Nicolaescu, B. Salamat, A. Veidenbaum and M. Valero, 'Fast Speculative Address Generation and Way Caching for Reducing L1 Data Cache Energy,' in *2006 International Conference on Computer Design*, ISSN: 1063-6404, Oct. 2006, pp. 101–107. DOI: 10.1109/ICCD.2006.4380801.

[28] G. S. Sohi and M. Franklin, 'High-bandwidth data memory systems for superscalar processors,' in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS IV, Apr. 1991, pp. 53–62.

[29] S. Mittal, 'A Survey of Recent Prefetching Techniques for Processor Caches,' *ACM Computing Surveys*, vol. 49, no. 2, 35:1–35:35, Aug. 2016.

[30] C. Celio, D. Patterson and K. Asanovic, 'The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor,' en, Electrical Engineering and Computer Sciences University of California at Berkeley, Technical report UCB/EECS-2015-167, Jun. 2015.

[31] C. Celio, P.-F. Chiu, K. Asanović, B. Nikolić and D. Patterson, 'BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS,' *IEEE Micro*, vol. 39, no. 2, pp. 52–60, Mar. 2019.

[32] 'SiFive Tilelink Specification version 1.8.1,' SiFive Inc, Technical report, Jan. 2020. [Online]. Available: https://starfivetech.com/uploads/tilelink_spec_1.8.1.pdf.

[33] D. Biancolin, S. Karandikar, D. Kim, J. Koenig, A. Waterman, J. Bachrach and K. Asanovic, 'FASED: FPGA-Accelerated Simulation and Evaluation of DRAM,' in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2019, pp. 330–339.

[34] A. Agarwal, R. Simoni, J. Hennessy and M. Horowitz, 'An evaluation of directory schemes for cache coherence,' *ACM SIGARCH Computer Architecture News*, vol. 16, no. 2, pp. 280–298, May 1988.

[35] B. Gottschall, M. Jahre and E. Lieven, 'TEA: Time-Proportional Event Analysis,' in *To appear in the Proceedings of the International Symposium on Computer Architecture ISCA*.

[36] Z. Zhang, Z. Zhu and X. Zhang, 'A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality,' in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, Dec. 2000, pp. 32–41.

Chapter 9

Appendix

9.1 Memory benchmarks

Example of the main loops in the benchmarks used to test load and store bandwidth to the CPU

```
start:
li a5, mem_region_base
start_iteration:
    ld a4, 0(a5)
    ld a4, 8(a5)
    ld a4, 16(a5)
    ld a4, 24(a5)
    ld a4, 32(a5)
    ld a4, 40(a5)
    ld a4, 48(a5)
    ld a4, 56(a5)
    addi a5, a5, 64
    bne a5, a3, start_iteration // Run until entire memory region
↪   has been loaded from
    addi a2, 1
    bne a2, N_Iterations, start
```

Listing 1: Assembly code illustrating the streaming load microbenchmark used

```
start:
li a5, mem_region_base
start_iteration:
    ld a4, 0(a5)
    ld a4, 64(a5)
    ld a4, 128(a5)
    ld a4, 192(a5)
    ld a4, 256(a5)
    ld a4, 320(a5)
    ld a4, 384(a5)
    ld a4, 448(a5)
    addi a5, a5, 512
    bne a5, a3, start_iteration // Run until entire memory region
 ↪  has been loaded from
    addi a2, 1
    bne a2, N_Iterations, start
```

Listing 2: Assembly code illustrating the strided load microbenchmark used

```
start:
li a5, mem_region_base
start_iteration:
    sd a4, 0(a5)
    sd a4, 8(a5)
    sd a4, 16(a5)
    sd a4, 24(a5)
    sd a4, 32(a5)
    sd a4, 40(a5)
    sd a4, 48(a5)
    sd a4, 56(a5)
    addi a5, a5, 64
    bne a5, a3, start_iteration // Run until entire memory region
 ↪  has been loaded from
    addi a2, 1
    bne a2, N_Iterations, start
```

Listing 3: Assembly code illustrating the streaming store microbenchmark used

```
start:
li a5, mem_region_base
start_iteration:
    sd a4, 0(a5)
    sd a4, 64(a5)
    sd a4, 128(a5)
    sd a4, 192(a5)
    sd a4, 256(a5)
    sd a4, 320(a5)
    sd a4, 384(a5)
    sd a4, 448(a5)
    addi a5, a5, 512
    bne a5, a3, start_iteration // Run until entire memory region
 ↪  has been loaded from
    addi a2, 1
    bne a2, N_Iterations, start
```

Listing 4: Assembly code illustrating the strided store microbenchmark used

```
start:
li a5, mem_region1_base
li a6, mem_region2_base
start_iteration:
    ld a4, 0(a5)
    sd a4, 0(a6)
    ld a4, 8(a5)
    sd a4, 8(a6)
    ld a4, 16(a5)
    sd a4, 16(a6)
    ld a4, 24(a5)
    sd a4, 24(a6)
    ld a4, 32(a5)
    sd a4, 32(a6)
    ld a4, 40(a5)
    sd a4, 40(a6)
    ld a4, 48(a5)
    sd a4, 48(a6)
    ld a4, 56(a5)
    sd a4, 56(a6)
    addi a5, a5, 64
    bne a5, a3, start_iteration // Run until entire memory region
 ↪  has been loaded from
    addi a2, 1
    bne a2, N_Iterations, start
```

Listing 5: Assembly code illustrating the streaming loadstore microbenchmark used