Microbenchmarking Intel Knights Landing

Alexander Wilson

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2018

Abstract

This project investigates how processors with complex memory hierarchies and execution pipelines process instructions and data. This includes evaluating the performance of the memory system of Intel's 2nd Generation Xeon Phi Processor, codenamed Knights Landing. The report covers how instruction latency is measured and how important computer architecture concepts such as pipelining and out-of-order execution are accounted for when conducting measurements. It also covers usage of STREAM, an open-source benchmark for measuring memory bandwidth, on Knights Landing and presents results for latencies and bandwidths of memory components of the Knights Landing Processor. Finally it provides usage instructions for repeating the experiments and running the benchmark suite implemented.

The Results and accompanying Discussion also verify performance metrics provided by Intel on the MCDRAM and DRAM Bandwidths, that peak bandwidth of MCDRAM is 400+GB/s.

Acknowledgements

I would like to thank my supervisor **Vijay Nagarajan** for his help in supervising this project. Not only did he help me throughout the project, but his teaching in the Computer/Parallel Architecture courses helped me understand fundamental concepts in order to work on this project.

I would also like to thank **all my family** for their extensive support for my education, giving me the time and space I needed to study, and ultimately supporting me pursuing my interests.

Finally, I would like to thank my best friend **Eddy** for always being there to listen to me try to explain my work and being supportive.

Table of Contents

1 Introduction **3**
- 1.1 Project Motivation . . . 3
- 1.2 Project Overview . . . 3
- 1.3 Project Goals Achieved . . . 4
- 1.4 Report Structure . . . 4

2 Background **7**
- 2.1 Computer Architecture . . . 7
  - 2.1.1 Pipelining . . . 7
  - 2.1.2 Out-Of-Order Execution . . . 8
  - 2.1.3 Caching . . . 8
- 2.2 Parallel Architectures . . . 11
  - 2.2.1 Cache Coherence . . . 11
  - 2.2.2 MESI Protocol . . . 11
  - 2.2.3 MESIF Protocol . . . 13

3 Understanding KNL Architecture **15**
- 3.1 Chip Layout . . . 15
  - 3.1.1 Tile . . . 16
  - 3.1.2 L2 Cache . . . 16
  - 3.1.3 MCDRAM . . . 17
- 3.2 Threading . . . 17

4 Writing the Microbenchmarks **19**
- 4.1 Measuring Latency . . . 19
  - 4.1.1 Hardware Support/Tools . . . 19
  - 4.1.2 Important Considerations . . . 20
  - 4.1.3 Timing Technique . . . 22
  - 4.1.4 Generating Loads . . . 26
  - 4.1.5 Conclusion . . . 27
- 4.2 Measuring Bandwidth . . . 29
  - 4.2.1 Important Considerations . . . 29
  - 4.2.2 STREAM [14] . . . 29
- 4.3 L1 Cache Latency . . . 30
  - 4.3.1 Algorithm . . . 30
  - 4.3.2 Key Features/Integral Components . . . 31

4.4 L2 Cache Latency . . . 33
4.4.1 Algorithm . . . 33
4.4.2 Key Features/Integral Components . . . 34
4.5 DRAM/MCDRAM Latency . . . 35
4.5.1 Algorithm . . . 35
4.5.2 Key Features/Integral Components . . . 36
4.6 Remote Cache Latency(Coherence Miss Latencies) . . . 37
4.6.1 Thread Structure . . . 37
4.6.2 Thread Tasks . . . 38
4.6.3 Remote L1 Modified Read . . . 39
4.6.4 Remote L1 Exclusive Read . . . 41
4.6.5 Remote L1 Shared Read . . . 43
4.7 DRAM/MCDRAM Bandwidth . . . 45
4.7.1 STREAM Benchmark . . . 45
4.7.2 STREAM Usage . . . 45

5 Results 47
5.1 Latencies . . . 47
5.1.1 Intra-Tile Latencies . . . 47
5.1.2 Inter-Tile Latencies . . . 47
5.1.3 MCDRAM/DRAM Latencies . . . 48
5.2 Bandwidths . . . 49
5.2.1 All-to-All . . . 49
5.2.2 Quadrant . . . 50
5.2.3 Sub-NUMA . . . 51

6 Discussion 55
6.1 Latency Conclusions . . . 55
6.1.1 Intra-Tile Results . . . 55
6.1.2 Inter-Tile Results . . . 57
6.1.3 MCDRAM/DRAM Latencies Results . . . 57
6.2 MCDRAM/DRAM Bandwidth Results . . . 58
6.3 Summary . . . 59

7 Conclusion 61
7.1 Work Done . . . 61
7.2 Insight Gained . . . 62
7.3 Project Outcome . . . 62

8 Open-Source Benchmark Suite 63
8.1 Repository Structure . . . 63
8.2 Benchmarks . . . 63
8.2.1 Coherence Miss . . . 64
8.2.2 Latencies . . . 64
8.2.3 Memory Bandwidth . . . 64
8.2.4 Tools . . . 65
8.3 Docs . . . 65

Bibliography **67**

A Appendices **69**

A.1 Timestamp Counter Support 69

A.2 Results 70

Chapter 1

Introduction

1.1 Project Motivation

With processors becoming increasingly complex, and the need for large data processing, hardware manufacturers like Intel are producing new and different processors to tackle different problems, using new or different architectural designs. One of the major issues with modern processors is that the memory system is a huge bottleneck in the processor; for example, fetching data from memory takes substantially longer than most arithmetic or logical operations. In order to tackle this issue, complex memory hierarchies of Caches and DRAM are created to try and minimise the latency of a memory access. It is interesting to investigate how processors work, and how these complex memory systems behave, in order to understand how these systems improve program performance.

1.2 Project Overview

The goal of this project is to evaluate the performance of Intel's 2nd Generation Xeon Phi processor's(code-named Knights Landing, or KNL for short) memory system, in order to verify Intel's performance statistics. The project is also about gaining an understanding of how High Performance Computers differ to desktop processors, and an understanding of how parallel processing is achieved in a practical sense. Knights Landing has a complex and interesting memory system, including new memory modules relative to its predecessors. The project scope included measuring the latencies of accessing different regions of memory, namely the L1 Cache, L2 Cache, DRAM, and Intel's new MCDRAM. The project scope also includes investigating how the Cache Coherence system present across the chip behaves, and what the latency of accessing data between Cores is in the different Coherence States. The project scope also includes measuring the bandwidth of the DRAM and MCDRAM, to better understand how data is processed in a High Performance Computing context.

1.3 Project Goals Achieved

From the goals briefly outlined in the Project Overview concrete results were obtained for the Bandwidth's of the DRAM and MCDRAM. Concrete results for the latencies of the L1 Cache, L2 Cache, DRAM and MCDRAM. However, the accuracy and validity of the results will be discussed in the Discussion Chapter. Finally, concrete results for the latencies of accessing data in remote Cores was partially obtained for some states, but more work would need to be done to validate the results or obtain other reliable results.

1.4 Report Structure

Following the Introduction;

Chapter 2 will cover background information on computer architecture that will give a general understanding of how processors execute instructions and tackle the long latencies of DRAM accesses using Caches. The Background will also include some rudimentary information on how parallel processors work, and keep their Cache's data coherent across Cores.

Chapter 3 will then outline the architecture of the Knights Landing Processor, including the configurable modes for the L2 Cache, and the MCDRAM. The Chapter will also lightly discuss how the KNL Cores handle Threads, which will be relevant when discussing the measured results.

Chapter 4 is made up of 7 sections:

1. Measuring Latency
2. Measuring Bandwidth
3. L1 Cache Latency
4. L2 Cache Latency
5. DRAM/MCDRAM Latency
6. Remote Cache Latency(Coherence Miss Latencies)
7. DRAM/MCDRAM Bandwidth

Section 4.1 discusses the work done in a chronological order, for deciding upon a technique for measuring instruction latency. This section also includes practices used to verify techniques, and the methodology for generating Load instructions. The conclusion of this section is a portable methodology for generating Load instructions, and an approach to measuring instruction execution latency.

Section 4.2 briefly introduces the Open Source benchmark program that was used for measuring the bandwidth of the DRAM and MCDRAM.

Section 4.3 discusses the algorithm of the benchmark program for measuring the latency of an L1 Cache load, which includes a high-level overview of the algorithm, using the conclusions of Section 4.1(Measuring Latency).

Section 4.4 discusses the algorithm of the benchmark program for measuring the latency of an L2 Cache load, which includes a high-level overview of the algorithm, using the conclusions of Section 4.1(Measuring Latency). This section will build upon Section 4.3 and note important differences.

Section 4.5 discusses the algorithm of the benchmark program for measuring the latency of a DRAM/MCDRAM load, again this will build upon the previous two sections, noting important differences.

Section 4.6 details how I implemented a multi-threaded benchmark application. It also shows how information highlighted in the Background Chapter (Section 2.2.1) is used to place the Processor into a particular state for measuring latencies for accessing data across Cores(into other Caches).

Finally, Section 4.7 discusses what the Bandwidth microbenchmark does, and why it is relevant. This includes discussion of how it was used to ensure the results obtained are valid.

Chapter 5 presents the results of running the microbenchmarks from Chapter 4.

Chapter 6 discusses the results obtained, and the validity of the results compared to Intel's published results and results for similar architectures. This Chapter also includes discussion on what could be done to improve the results.

Chapter 7 concludes the report, highlighting what work was done, including what was learnt throughout the project. This Chapter will finish by concluding the outcome of the project, including deliverables and further work.

Chapter 8 is separate to the main flow of the report, and introduces the Open Source version of the implemented Benchmark Suite, including instructions for usage.

Chapter 2

Background

2.1 Computer Architecture

This section will cover design features present in modern processors that are relevant to the scope of this project, specifically in understanding how benchmarks are written.

2.1.1 Pipelining

Pipelining is a technique used in almost all modern processors to commit more instructions per cycle. The technique involves splitting an instruction into stages, and then once any given stage has completed, a new instruction can enter that stage. For example, we could consider a processor to process instructions over 5 stages:

1. **`IF` - Instruction Fetch**
   The instruction at the program counter(PC) is fetched from memory, and the PC is incremented.
2. **`ID` - Instruction Decode**
   The fetched instruction is decoded, and required values are fetched from general purpose Registers.
3. **`EXE` - Execution**
   The arithmetic and logic operations are computed. This includes computation of addresses.
4. **`MEM` - Memory Access/Branch Completion**
   Memory is accessed if required.
5. **`WB` - Write Back**
   Results of execution are written back to general purpose Registers

Splitting instruction execution into stages, like the above model, allows the processor to reallocate a given stages resources to a new instruction, once the current instruction has finished using those resources. For example, the processor can begin fetching the next instruction after it has finished fetching the current instruction, but before the current instruction has totally finished being executed. This means that in the ideal situation, our model can have 5 instructions in flight, and given that each stage costs one cycle, potentially committing an instruction every cycle. It is important to note that in practice, this is likely won't happen, for example, complex arithmetic such as divisions will take multiple cycles to complete the `EXE` stage.

2.1.2 Out-Of-Order Execution

Out of order execution is a technique used in processors to increase the efficiency of the pipeline. The processor can re-order instructions to make more efficient usage of the different stages. For example, if you have a complex division instruction, followed by a few simple addition instructions, the processor could push the additions into the pipeline first, and the division last, so that the pipeline doesn't halt or bottleneck at the `EXE` stage while waiting for the complex division to be calculated, leaving other instructions halted in flight behind it. Out-Of-Order execution is complex, and if the CPU were to re-order dependent instructions, it would have other systems in place to handle instruction dependencies. An example of such a dependency would be:

```
x = a / b
y = x + 5
```

Where the value of `y` cannot be calculated until the result of `x` has been calculated, and so `y = x + 5` may hang at the `ID` stage, essentially halting the pipeline. However pipelining techniques to tackle instruction dependencies are out of the scope of the project, but it is important to understand that modern processors may re-order instructions.

2.1.3 Caching

2.1.3.1 Motivation

Moore's Law states that the number of transistors that could be fit onto a computer chip would scale up by a factor of five, every five years. To begin with, the extra transistors on the chips were used for created logical units that could compute logical/arithmetic operations quicker, however it was soon realised that despite processors being able to process arithmetic with a latency as low as a single cycle, actually retrieving the data to perform the logical/arithmetic operations on was becoming a huge bottleneck. For example; the `EXE` stage of the pipeline could process simple addition operations in a single cycle, but retrieving the additions operands can cost up to a few hundred cycles to retrieve from memory, and so memory latency became a significant bottleneck in the pipeline.

2.1.3.2 Solution

The solution to this problem has not just been to throw more hardware at main memory, but to use the new transistors on chips to create a memory hierarchy that caches and pre-fetches data that is likely to be used. Most general purpose CPU's today have a two to three level cache hierarchy, with level one being the smallest and closest to the processor cores, and level 3 being the biggest. When referring to different levels of caches in a processor, `L1` is used to refer to the level one cache; likewise `L2` to refer to the level two cache.

2.1.3.3 Cache Behaviour

Each processor model will have its own unique cache architecture. However it is typical for an `L1` cache to exist spatially close to a single core, and be unique to that core. Some architectures have `L2` caches unique to a core, however, it is more common that the `L2` cache is shared among cores. There are two ways that caches are designed to be indexed; Direct-Mapping and Set-Associative.

- **Direct Mapping**
  Each line in memory is mapped to a specific line in the cache. If memory has `M` lines, and our cache has `C` lines, where `C < M`, memory line `m` is located in the cache at line `m modulo C`
  This has the advantage of being quick to access, as you simply look up the line in the cache where the address you wish to read/write to would exist, and if the tag of the cache-line and your address match, you have found your data quickly. This method can incur cache thrashing. Cache thrashing is when useful data is evicted from the cache. For example, say you have two variables in memory, and they both map to the same cache line, the cache could be evicting and replacing this line potentially many times depending on the nature of the program and its use of program variables/memory.

  Figure 2.1, shows how lines in memory are mapped in a cyclic fashion to lines in the cache.

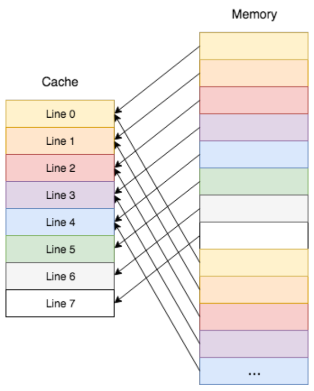


Figure 2.1: Direct Mapped Cache Diagram

- **Set Associative**
  An N-way set associative cache is a cache that is split into sets containing N lines. Therefore if we have a cache with 32 lines, and it is 4-way set associative, then our cache has 8 sets, each containing 4 lines. Memory is mapped to set-associative caches similarly to direct mapping, but instead of mapping a line in memory to a specific line in a cache, it is mapped to a set in a cache, and then the contents of the sets are managed using a cache replacement policy(such as least-recently-used (LRU), first-in-first-out or random-replacement). This also allows for a fully-associative cache, where there is only 1 set, where all cache lines are stored/evicted based on a replacement policy.
  This has the advantage of dealing with cache trashing, where multiple lines can exist in the cache at the same time, that would not exist simultaneously in a Direct Mapped cache.
  In the case of a fully-associative cache, there is a large overhead in searching the cache for the data that you want, and so in practice, N-way set associative caches are often used in general purpose CPU's to get the best of both Direct Mapped and Fully Associative mappings. But that is not to say that there are no use-cases for Direct Mapped caches.

  Figure 2.2, shows how lines in memory are mapped to one **set** in the cache. It's important to note that the items within a **set** are evicted based on the architectures chosen replacement policy.

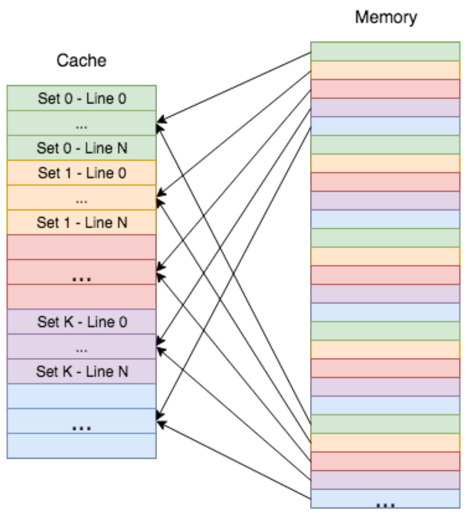


Figure 2.2: N-way Set Associative Cache Diagram

It is also important to note that a cache line can vary in size depending on architecture, but it is common for cache lines to be 64B, as it is affordable to transport 64B between caches and memory while exploiting program spatial locality[1].

[1]Program Spatial Locality: Programs often access nearby memory addresses. For example, incremented indexes in an array.

2.2 Parallel Architectures

2.2.1 Cache Coherence

Modern Processors are often Multi-Core systems, with each Core having its own Cache System. Figure 2.3 illustrates a simple Chip which may consist of 4 Cores connected to a Bus and DRAM.

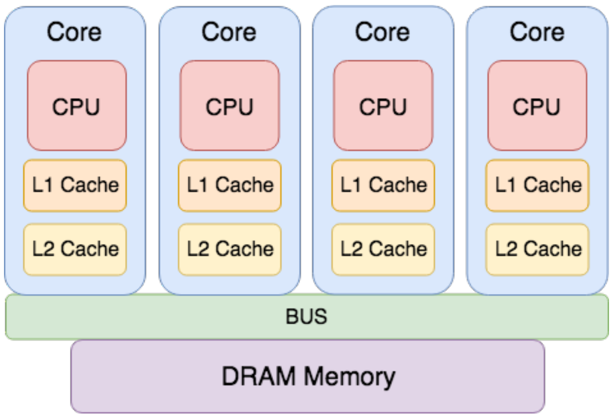


Figure 2.3: Parallel Chip System

It is easy to envision that programs would exist where data can be processed in a parallel fashion; i.e. where the majority of the processing does not need to be done in any particular order. Such a scenario would mean that all those processing tasks can be run simultaneously on different cores. Cache Coherence Protocols are a means of tracking where data exists in the different Cores, and ensuring that only one **value** of a particular piece of data exists. In practice, Coherence Protocols offer write serialisation[2] and write-propagation[3] in many-core systems, as they essentially abstract away the existence of multiple Caches, and create a system where there appears to be only one large Cache. Coherence Protocols are implemented using States, which track where the data exists, if it has been modified, and if it's valid.

2.2.2 MESI Protocol

The `MESI` Protocol is a Cache Coherence protocol that is based on the following four states:

- **Modified**
  A line that exists in Modified State in a Cache guarantees that it is the only valid version of this Line across all Cores, where the Line is dirty[4].

[2]Write-Serialisation: The notion of all cores in a system mutually agreeing on the order of variable writes.
[3]Write-Propogation: The notion that the outcome of the write must eventually reach other cores.
[4]Dirty Line: A line which has been modified and its value not written back to memory yet.

- **Exclusive**
  A line that exists in Exclusive State in a Cache guarantees that it is the only valid version of this Line across all Cores, where the Line is NOT dirty.
- **Shared**
  A line that exists in Shared State in a Cache guarantees that there *may* exist other unmodified copies of this Line in one or more of the other Cores.
- **Invalid**
  A line can exist in the Invalid State in a Cache, meaning that its value has been invalidated. A line can be invalidated by another Core modifying a line, which invalidates other copies of the same Line.

In the most simple case, these protocols are enforced over the means of ”snooping” the bus medium. This means that there are controllers on each Core that are constantly listening on the Bus, can send messages on the Bus, and react to messages sent on the Bus. Figure 2.4 shows the State Diagram of the `MESI` Protocol. Although the State Diagram is quite complex, it is important to note that upon a **Remote Core** performing a Write, all other Cores Invalidate their Line, and that regardless of State, if a **Local Core** performs a Write, it always enters the Modified State. This shows that only one value per data can exist in a Coherent Cache System.

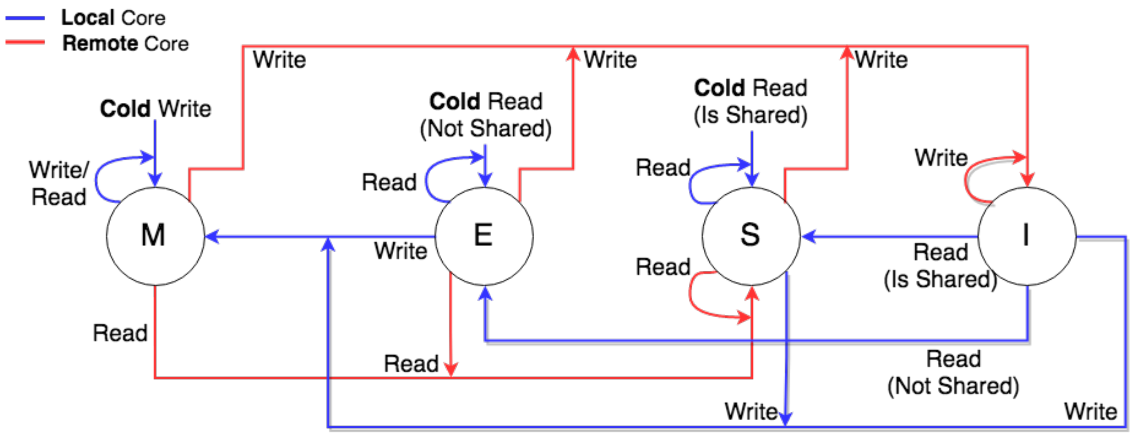


Figure 2.4: `MESI` Protocol - State Diagram

An example of how the State Diagram in Figure 2.4 can be interpreted:

Say we have 4 Processors: [P0, P1, P2, P3], and we use the following format to denote P1 performing a Read to Line 5: `P1 R 5`, and P3 performing a Write to Line 1000: `P3 W 1000`. If we want to generate a scenario where Processors 0 and 1 both own line 50 in the Shared State, we can do the following:

```
P0 R 50
P3 R 50
```

Using Figure 2.4, we look at P0, it performs a ”Cold Read” to Line 50, and so now P0 owns Line 50 in Exclusive State. When P3 performs a ”Cold Read” to Line 50, the State transitioning from ”**Cold** Read (Is Shared)” triggers, and P3 now owns Line

50 in the Shared State. Because this Read is communicated via the bus, it would, in fact, be P0 that supplies P3 with the Line, and P0 would observe a ”Remote Read”, which following the State Diagram, changes P0’s Exclusive Line 50, to a Shared Line 50. Now both P0 and P3 own Line 50 in the Shared state. Later sections that discuss Coherence Latencies will make reference to the `MESI` and `MESIF` Protocols in order to get data into certain states in a benchmark setting.

2.2.3 MESIF Protocol

The `MESIF` Protocol is an extension and modification of the `MESI` protocol, that adds the new Forward State, and alters the purpose of the Shared State. In `MESIF`, a line that exists in Shared State cannot be copied, and instead, it holds that a single instance of the Shared Line will be held in the Forward State[6], and that the Core that holds the Line in Forward State shall be responsible for forwarding the Data. There is also no guarantee that the Line in Forward State will be spatially close to the requesting Core. The motivation for this additional Forward state, is to reduce traffic on the Bus, because in MESI, if a Core requests data that exists in multiple other Cores in Shared State, all the Cores will respond to the request. However, in the MESIF protocol, the Core that contains the Shared Line in Forward State is the only Core responsible for responding to the Request. This is enough of an understanding of the `MESIF` Protocol that is required for the scope of this project, as we will only be interested in setting up Cores to own lines in different Coherence States.

Chapter 3

Understanding KNL Architecture

This chapter will outline the KNL Architecture, detailing what needs to be known to understand how the benchmarks work.

3.1 Chip Layout

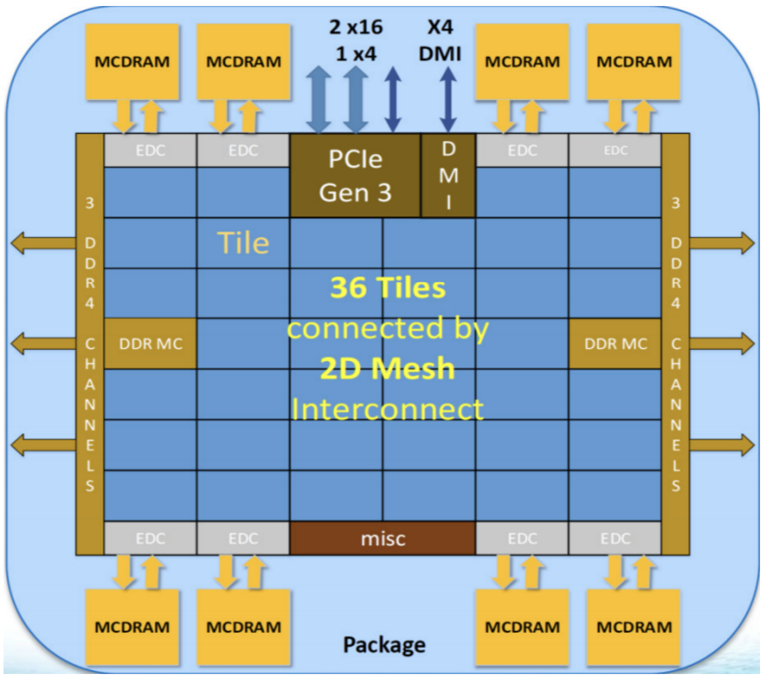


Figure 3.1: KNL Chip Layout
[17]

Figure 3.1, shows a top-down view of the architecture of the KNL chip. The chip is a mesh of 36 tiles.

3.1.1 Tile

A tile consists of:

- 2 x CPU Cores:
  - `32KB L1` Instruction Cache `8-way` set-associative, `64B` cache lines
  - `32KB L1` Data Cache `8-way` set-associative, `64B` cache lines
- 1 x L2 Cache
  - `1MB` Cache shared between Cores.
  - `16-way` set-associative, `64B` cache lines
  - Performs a single `read` in `1 cycle`.
  - Performs a single `write` in `2 cycles`.
  - L2 Cache **coherent** with all other tiles L2 Caches.
- 1 x Caching/Home Agent
  - Distributed Tag Directory to keep `L2 Caches` coherent across tiles.

3.1.2 L2 Cache

The mesh of tiles creates a distributed `L2` Cache. This mesh is configurable in three different modes:

- **All-to-All**
  All of the `L2` Address space is uniformly distributed across all tiles. This means that any given core can access data in the `L2` Cache of any other tile. A typical miss in All-to-All mode starts with a Core missing in its `L1` Cache, it then checks its own tiles `L2` Cache, and upon missing, checks the **distributed** directory. After missing in the **distributed** directory, the request is forwarded to MCDRAM/DRAM(Depending on configuration).
  Most general mode, so relatively lower performance.
- **Quadrant**
  The Chip is divided into four virtual quadrants. Addresses are hashed to a directory in the same quadrant. This means that a core can access data in the `L2` Cache of any other tile in the same virtual quadrant. A typical miss in Quadrant mode starts with a Core missing in its `L1` Cache, it then checks its own tiles `L2` Cache, and upon missing, checks the **quadrant** directory. After missing in the **quadrant** directory, the request is forwarded to MCDRAM/DRAM(Depending on configuration).
  This mode has a lower latency, and higher bandwidth relative to `All-to-All`. Quadrants are transparent to software.

- **Sub-NUMA Clustering**
  The Chip is divided into four quadrants. Each exposed to the OS as a separate NUMA[1] domain. To the operating system, this looks analogous to a 4 socket Xeon(server) processor. A typical miss in Sub-NUMA mode starts with a Core missing in its `L1` Cache, it then checks the **quadrant** directory. After missing in the **quadrant** directory, the request is forwarded to MCDRAM/DRAM(Depending on configuration).
  This mode has the lowest latency of all three modes, due to the nature of NUMA and spatial locality, but requires software support to handle the distributed memory.

3.1.3 MCDRAM

The KNL Chip includes a new layer of high-bandwidth memory that is spatially close to the chip, but has a higher access latency compared to DRAM. Intel coined MCDRAM is 16GB of DRAM built into the chip, with three configurable modes:

- **Cache Mode**
  The 16GB of memory is used to cache DRAM. This is completely transparent to software, and so any memory address that doesn't exist in the `L1` or `L2` Cache's is directed to MCDRAM next. When considering access latencies, visiting MCDRAM is already more costly than visiting DRAM, but missing in MCDRAM also then requires a further look up to main memory. This effectively means that Cache Mode is best used for frequently used contiguous[2] memory, as data can be pulled into `L2` and `L1` Caches with fewer accesses to memory.
- **Flat Mode**
  The 16GB of memory is used to extend DRAM, essentially extending its address space. This mode requires programs to be aware of this configuration, and address memory appropriately
- **Hybrid Mode**
  The 16GB of memory can be used to both extend DRAM, and cache it. It can be configured to use either 25% or 50% of the 16GB as a DRAM Cache, with the rest extending DRAM. This has the benefit of extending DRAM, and also caching data that is frequently used.

3.2 Threading

Intel's Presentation[17] on the Knights Landing chip highlights the Core's multithreading capabilities. It is important to note that Core's resources are shared or dynamically re-partitioned among threads, so stalls will be incurred when thread contexts

[1]NUMA: Non-Uniform Memory Access; The idea that different cores have their own physical memory.

[2]Contiguous Memory: Memory that exists spatially next to each other.

are switched. These observations will be particularly noticeable when discussing our results.

Chapter 4

Writing the Microbenchmarks

This chapter will talk about the different approaches to benchmarking latencies and bandwidths of different memory components, and tackling obstacles that would prevent accurate or reliable results. It will then describe the how the discoveries made were used to create microbenchmarks for the L1 Cache, L2 Cache, DRAM, and MC-DRAM.

4.1 Measuring Latency

For measuring latencies, a reliable timing technique must be used that is accurate enough to time fine-grain operations to nanosecond precision. At a high-level, the latency measuring technique involves taking a reading of time before and after an instruction(s) of interest, and calculating the difference between said times, minus the overhead of timing. It is also important to make sure that we are timing the desired instructions.

4.1.1 Hardware Support/Tools

In order to perform high-precision latency benchmarks, we must have hardware support and tools that allow us to take fine-grain measurements of time, as standard c++ timing libraries cannot provide the precision required to gain any meaningful results.

4.1.1.1 Timestamp Counter (TSC)

The Timestamp Counter (TSC) is a measuring ”device” that allows us to get information on the current cycle-count of the processor. Traditionally, the TSC was a register that existed in each Core. The value in the register would be incremented every cycle, and therefore could be read and the value returned could be used to count the number of cycles elapsed. This was useful for programmers to benchmark how their code would perform in terms of cycles spent, but would not provide a portable solution for measuring wall-time[1] since wall-time is a combination of cycles spent and core frequency.

Calculating the wall-time using the traditional TSC becomes increasingly difficult when we consider that Intel uses technologies to change Core's frequencies dynamically (for purposes such as power-saving). Since the TSC is also local to each individual Core, it cannot be used to measure code that runs concurrently across multiple cores. Modern Intel CPUs provide access to an ”invariant TSC”, which increments at set intervals, meaning it can be used for measuring wall-time. Knights Landing does have an invariant TSC, and as such can be used in our microbenchmarks.

Refer to Appendix A.1 for information on what your Processor's TSC support is.

4.1.1.2 Assembly Instructions/Flags

In order to make use of our hardware tools, we must have appropriate Assembly Instructions. The three Instructions of interest to this project are:

- **`CPUID`[18]**
  This instruction is a serialising instruction, which guarantees:
  ”any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.”
- **`MFENCE`[19]**
  Assembly Instruction/Flag which guarantees that all loads/stores prior to it have been globally retired. Meaning that the resulting memory operation is visible globally.
- **`RDTSCP`[20]**
  Hardware supports a call to reading the TSC that waits until all previous instructions have finished executing and are globally retired via the `RDTSCP` Instruction. But is NOT a serialising Instruction.

As the techniques for timing are explored we will see the relevance of all the above Assembly Instructions.

4.1.2 Important Considerations

Despite having good hardware tools, there are still some issues that must be considered and tackled in order for our chosen timing technique to be valid.

The three core considerations are:

1. Core Affinity: Process/Thread Placement
2. Compiler Optimisations

[1]Wall-Time: Real-world time.

3. Out-of-Order Execution

4.1.2.1 Core Affinity

Core affinity is the notion of binding a Process or Thread to a Core or range of Cores on a system. This is an important consideration as the TSC is Core Specific. Therefore we want to make sure that we pin our threads to a single core, to ensure that our TSC reads are reading the same register, as there is no guarantee that all TSC Registers across cores will have the same value. It also prevents us accidentally measuring the latency of migrating a process/thread to another Core.

The Linux c++ library `sched.h` provides a library for scheduling, and StackOverflow user Ankur Chauhan provided a clean implementation[4] for pinning the current thread to a specific Core using `sched.h`. The implementation was tested to ensure it works appropriately and subsequently it was used within the microbenchmarks.

4.1.2.2 Compiler Optimisations

All compilation is done using the following command to ensure that the features of c++ we require are available:

```
prog.cpp -std=c++0x [-pthread] -o coherence_latencies.o
```

It is important to consider optimisations that the Compiler may use when generating the executable. To ensure that critical sections of the microbenchmarks are not optimised out, we can make use of the `volatile` keyword, which tells the compiler not to remove the code, in the event that it detects it to be dead code. Later sections will show how the **volatile** keyword is used to ensure code is not removed.

4.1.2.3 Out-of-Order Execution

Out-of-Order Execution (OoOE) was introduced in Section 4.1.2.3, and is a major obstacle in writing correct microbenchmarks. If the microbenchmark code is not crafted carefully, instructions will be shuffled around by the processor to allow the code to stream through the pipeline quicker, and the instructions that the microbenchmark intends to time, may not be the timed instructions. We will see in Section 4.1.3 how the development of the timing component of our microbenchmarks takes OoOE into account.

As I discuss the methodology for arriving at the chosen timing technique, I will refer to these considerations and how they affected the chosen technique.

4.1.3 Timing Technique

This section will progressively describe the chosen timing technique with reference to the considerations in Section 4.1.2, and how the timing technique was verified.

4.1.3.1 Timing Verification

In order to have a timing mechanism that could be reliably used in a microbenchmark, it was important not just to experiment with different timing techniques, but to be able to verify our experiments using known values.

Using Agner Fog's "Instruction Tables"[7], which details cycle latency for standard x86 Instructions on different CPU Architectures, a test suite coined the "Sanity Tests" was crafted to ensure that when using a given timing mechanism, it would return valid latencies. These Sanity Tests would also perform an "overhead" test to ensure that the overhead of the timing mechanism was consistent and reliable, otherwise, it would not be possible to subtract the overhead from our measured latency.

The Sanity Tests timed 1000 runs of the following instructions, and the returned latencies could be compared to Fog's table entry for the Knights Landing processor:

- **`DIV`**
  The Division Instruction was expected to have a latency of **29-95 Cycles**.
- **`PAUSE`**
  The PAUSE Instruction was expected to have a latency of **25 Cycles**
- **`F2XM1`**
  The F2XM1 Instruction was expected to have a latency of **100-400 Cycles**.

Sanity Test Algorithm:

```
latencies = [0] * 500
for i in range(0,1000):
    start = start_timestamp()
    # Timed Instruction, empty if timing Overhead.
    end   = end_timestamp()
    latencies[(end - start)]++

for idx, latency in enumerate(latencies):
    print str(idx) + " Cycles Occurred " + str(latency) + " Times"
```

4.1.3.2 BenchIT Solution[3]

The initial timing mechanism tested was based on the source code of the BenchIT microbenchmark suite:

```
unsigned long long timestamp()
{
  if (!has_rdtsc()) return 0;
  __asm__ __volatile__("rdtsc;": "=a" (reg_a), "=d" (reg_d));
  return (reg_d<<32)|(reg_a&0xffffffffULL);
}
```

This code produces a simple piece of assembly that reads from the TSC and stores the upper 32-bits into one register, and the lower 32-bits into another register, then returns a 64bit integer result. The BenchIT solution also includes a code snippet which essentially ”warms-up” the `RDTSC` instruction, which when included in the microbenchmarks, improved consistency of the results across runs.

This is a good foundation on which to build a timing mechanism but it does not inherently tackle the issue of `OoOE`, so it must be modified to include some serialising instruction(s).

4.1.3.3 Serialising via `CPUID`

Building upon the solution provided by BenchIT, experiments with placing `CPUID` Instructions around the `RDTSC` call were experimented with (including solutions provided in an Intel HPC Guide[16]) and run through the Sanity Tests, however, none of the experiments produced reliable, consistent or correct results.

4.1.3.4 Intel’s Benchmark Code Execution Time[15]

In further research, a paper[15] by Gabriele Paoloni deep dives an approach to benchmarking code execution time. The proposed timing technique for systems that support the `RDTSCP` instructions is as follows:

```
asm volatile (
    "CPUID\n\t"
    "RDTSC\n\t"
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t"
    :"=r" (cycles_high), "=r" (cycles_low)
    ::"%rax", "%rbx", "%rcx", "%rdx"
);
/* Code to time here - Critical Section */
asm volatile(
    "RDTSCP\n\t"
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t"
    "CPUID\n\t"
    :"=r" (cycles_high1),"=r" (cycles_low1)
16     ::"%rax", "%rbx", "%rcx", "%rdx"
17 );
```

An explanation of the code is as follows (also available on Pages 16-17 of [15]):

1. *Line 2* calls `CPUID` which ensures that any code prior to our Critical Section finishes execution, and cannot be executed within our Critical Section or Timing Mechanism.
2. *Line 3* calls `RDTSC`, which reads the TSC into the **eax** and **edx** registers.
3. *Line 4*, *Line 5*, and *Line 6* then store the **eax** and **edx** values to memory.
4. *Line 7* Informs the Compiler which registers are in use (clobbered) for its own internal knowledge.
5. *Line 9* will then contain the Critical Section of code which is to be benchmarked. It is advisable to have the code in-lined here in order to ensure that the overhead of a function call is not measured.
6. *Line 11* calls `RDTSCP` which ensures that all the previous instructions have finished executing, and then reads the TSC into the **eax** and **edx** registers.
7. *Line 12*, *Line 13*, and *Line 15* then store the **eax** and **edx** values to memory.
8. *Line 14* calls `CPUID` which ensures that any code prior has finished executing and that no proceeding code will be executed prior to *Line 14* which would poison our timing mechanism.
9. *Line 16* Informs the Compiler which registers are in use (clobbered) for its own internal knowledge.

This technique was run against the *Sanity Check* and produced the most consistent results of all techniques. And the measured latencies aligned with ballpark figures.

Table 4.1 shows the results of timing each of the following instructions: {`DIV`, `PAUSE`, `F2XM1`} 1,000 times, and counting the observed latencies in each run:

Table 4.1: *Sanity Check* Results - 1,000 Runs Per Instruction

Latency Observed	Overhead	DIV(29-95)	PAUSE(25)	F2XM1(100-400)
39 Cycles	822	0	0	0
52 Cycles	178	0	0	0
65 Cycles	0	0	847	0
78 Cycles	0	691	153	0
91 Cycles	0	309	0	0
377 Cycles	0	0	0	537
390 Cycles	0	0	0	462

As mentioned, Table 4.1 shows that we are getting consistent ballpark figures for our latency measurements, but not precise values. If we look purely at the set of Latencies observed, we can see that the latencies we observe are all multiples of 13. This indicates that our timing technique can only take a measurement every 13 Cycles. It can also be observed that there is always a portion of our observations (let us call it an anomaly) that is off by 1 measurement unit (13 Cycles being our ”measurement unit”). Regardless of how many trials we run for each instruction, be it 1,000,000 or 50; this anomaly always exists.

4.1.3.5 Further Analysis of Intel's Solution

With a benchmarking technique that provides consistent figures, further analysis had to be done to understand how we could improve the accuracy of the results.

As observed, the results were consistent but inaccurate. This was due to all observed latencies being multiples of 13 Cycles. It was hypothesised that this issue exists as a result of an alignment issue with our timing technique and the instruction being timed, and experiments were conducted with timing different numbers of the same Instruction consecutively. For example, instead of timing one single `PAUSE` Instruction, we could instead time `N` consecutive `PAUSE` Instructions and divide our latency (without the overhead) by `N`.

After trialling various values for the number of consecutive Instructions, it was found that 13 Instructions yielded results where the observed Latency was most similar to the actual Latency. Figure 4.1 shows a rough pictorial representation of the hypothesis/observation in action, where green blocks represent a 13 Cycle interval, and the blue blocks represent a 25 Cycle `PAUSE` instruction:

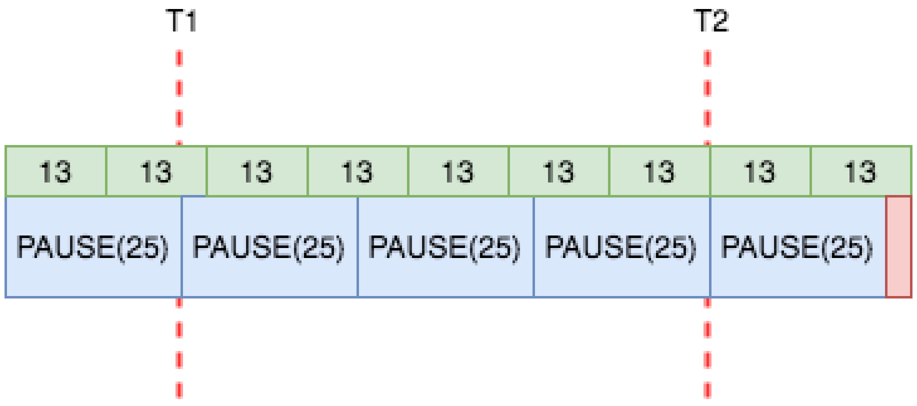


Figure 4.1: `PAUSE` Measuring Alignment

Although Figure 4.1 does not perfectly ratio the size of our 13 Cycle blocks to our 25 Cycle PAUSE Instructions, thinking about the concept shows that at time `T1`, our `PAUSE` Instruction that we are measuring has completed, but due to the accuracy of our timing mechanism, we mark the end of its completion as 26 Cycles (after the second 13 Cycle block). However, we see that after so many Instructions, that at time T2, the end of a 13 Cycle block aligns with the end of a `PAUSE` Instruction. It makes sense that after performing 13 consecutive `PAUSE` Instructions, that the end of the final `PAUSE` would align with the of a 13 Cycle block. And that in general, for an Instruction `X` with a consistent Latency of `Y` Cycles, queuing 13 of `X` consecutively will result in `Y` 13 Cycle blocks aligning with the final `X` Instruction.

We can use this fact to take measurements to an accuracy of 1 Cycle, simply by chaining multiple of the measured Instructions together in multiples of 13 within the timing measurements. Our revised *Sanity Test* now includes instructions with a lower expected latency, to test this hypothesis, and the results are as follows, with the overhead remaining the same at 39 Cycles:

Table 4.2: *Sanity Check* Revised Results - 1,000 Runs Per Instruction

Latency Observed	CWDE(1)	PAUSE(25)	CWD(7)
1 Cycles	1000	0	0
6 Cycles	0	0	312
7 Cycles	0	0	685
25 Cycles	0	813	0
26 Cycles	0	185	0

Table 4.2 shows that we get much more accurate results for each Instruction, however, in some Instruction's cases, we still have this anomaly that exists. This could be due to various reasons, each of which are difficult to verify. Some examples of reasons that were considered but not investigated are:

1. **Assembly to Micro-Op Translation**
   Despite writing Assembly Instructions, x86 Instructions are in fact broken down further into Micro-Operations, this is advantageous to the processor as it allows the CPU to re-order 'stages' of an Instruction. This is difficult to investigate as it is Instruction specific, and for the scope of this project, we are interested mostly in Load Instructions, for which there is not much documentation for.

2. **Pipeline Optimisations/Considerations**
   What Intel's processors do internally is proprietary information, and it is not possible to know exactly what is going on inside the Processor or pipeline, it may be that for some Instructions (for example `CWD`) in Table 4.2) that the Processor is able to perform an optimisation that can cut the Cycle time for multiple or consecutive identical Instructions.

4.1.4 Generating Loads

It is important to consider how we will generate Load Instructions. It would be possible to use a simple piece of `c++` code such as:

```
volatile int temp;
temp = data[idx];
```

To generate a Load to `data[idx]`, however, this would also generate a Store to `temp` which would be behaviour we do not wish to time when conducting Load Latency Benchmarks. Therefore we must use the following code snippet to generate a load instruction:

```
asm volatile (
    "\n\tmov %%eax, %0"
    "\n\tMFENCE"
    ::"r"(data[0]):
);
```

Figure 4.2: Load Instruction `c++` Inline Assembly

If we look at the Assembly generated by the compilation of Figure 4.2:

```
1 mov eax, DWORD PTR [rbp-262272]
2 mov %eax, eax
3 MFENCE
```

`Line 1` shows the Load Instruction that we will wish to time, which moves a Line from Memory into a Register, the **eax** (without the proceeding '%') indicates that an arbitrary register will be allocated. `Line 2` shows a `MOV` Instruction generated by the compiler that would move the result of `Line 1` into the `eax` Register. We do not wish to time this second `MOV` Instruction, but can account for it when calculating our overhead. And `Line 3` shows the `MFENCE` flag that will ensure the previous Loads globally retire before moving on. When conducting benchmarks involving Loads, each Load will be generated by using the `c++` code in Figure 4.2.

An example tutorial detailing the specifics of how inline assembly in Figure 4.2 works can be found on www.codeproject.com[10].

4.1.5 Conclusion

From the observations made, it seems fair to hypothesise that there is an uncontrollable margin for error that exists when conducting repeated experiments using Gabriele Paoloni's suggested technique[15]. His suggested solution definitely tackles the three most important considerations that were outlined in Section 4.1.2, and further analysis discovered more considerations with respect to the accuracy of the mechanism must be made when using the technique.

Therefore the implementation of the start and stop timing mechanisms are shown in Figure 4.3 and Figure 4.4 respectively. It is important to note that the functions take in a reference to a 32-bit integer, and store the upper and lower 32-bits of the TSC in the referenced variables, but do not convert the two variables to a 64-bit value here, as that would incur an overhead that would pollute the measurement, this will become more evident in Section 4.3.2 when discussing the usage of the timing functions. Both functions have the attributes:

```
inline __attribute__((always_inline)) volatile
```

Which ensure that where ever the function is called, the function is in-lined to that location in the source. This is *extremely* important as it prevents a new stack-frame being created for the function call, and in turn, avoids the overhead of the appropriate stack operations, and the overhead of calling the function. Finally, the microbenchmarks will use the code snippet in Figure 4.2 that was established in Section 4.1.4 for generating independent Load Instructions.

```c
inline __attribute__((always_inline)) volatile void start_timestamp(
    uint32_t *time_hi,
    uint32_t *time_lo)
{
    asm volatile (
        "CPUID\n\t"
        "RDTSC\n\t"
        "mov %%edx, %0\n\t"
        "mov %%eax, %1\n\t": "=r" (*time_hi), "=r" (*time_lo)::
        "%rax", "%rbx", "%rcx", "%rdx"
    );
}
```

Figure 4.3: start_timestamp function

```c
inline __attribute__((always_inline)) volatile void end_timestamp(
    uint32_t *time_hi,
    uint32_t *time_lo)
{
    asm volatile(
        "RDTSCP\n\t"
        "mov %%edx, %0\n\t"
        "mov %%eax, %1\n\t"
        "CPUID\n\t": "=r" (*time_hi), "=r" (*time_lo)::
        "%rax", "%rbx", "%rcx", "%rdx"
    );
}
```

Figure 4.4: end_timestamp function

4.2 Measuring Bandwidth

The scope of the project includes measuring the bandwidth between the aforementioned `MCDRAM` memory, but also the standard `DRAM` memory.

4.2.1 Important Considerations

In its purest form, bandwidth is the volume of data that can be transferred concurrently between two points. It is important to note that the `DRAM` and `MCDRAM` will behave differently depending on how they are configured. Therefore in light of the goals of this project, we want to measure the bandwidth of the `MCDRAM` and `DRAM` under all the different memory configurations with a varying number of actives threads.

4.2.2 STREAM [14]

STREAM is a memory bandwidth benchmark developed by John D. McCalpin, Ph.D. that "measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels."[11] It is an industry standard for measuring `DRAM` memory bandwidth, and using `numactl`[2][1] can be use to target either the `MCDRAM` or `DRAM`. Section 4.7 will cover how the STREAM benchmark is used to measure the `MCDRAM` and `DRAM` bandwidth.

4.3 L1 Cache Latency

Building upon the conclusion of Section 4.1, this section will discuss in detail how measuring the latency of a Load to a Local L1 Cache is benchmarked, including noteworthy design choices that were made to ensure that desired functionality was achieved.

4.3.1 Algorithm

4.3.1.1 Algorithm Pseudocode

```
# Returns the Latency of a single L1 Load
def getL1Latency(overhead):
    CorePin(0)                      # Pin Thread to Core 0
    data[L1_SIZE]                   # Array that will fill L1 Cache
    latencies[500] = {0,0,...,0}    # Init array of size 500 to all 0's
    for i in range(0,1000):
        for j in range(0, L1_SIZE):
            data[j] = j             # Write to data to get it in L1.

        start = start_timestamp()
        # Start Critical Section
        # Perform 26 Loads to data.
        load(data[0])
        asm("MFENCE")
        load(data[1])
        asm("MFENCE")
        ...
        load(data[25])
        asm("MFENCE")
        # End Critical Section
        end   = end_timestamp()

        latency = (end - start - overhead) / 26
        latencies[latency]++         # Increment count of this latency seen

    printLatencies(latencies)        # Prints results
```

Figure 4.5: L1 Latency Algorithm

4.3.1.2 Algorithm Walkthrough

1. Pins the Process/Thread to a single Core
2. Declares a data array that will be large enough to fill the L1 Cache.

3. Declares an array (of size 500), where the i'th element is the number of times a Load was measured to have taken i Cycles.
4. Loops for 1000 times, to perform repeated trials.
5. Loop through the L1 Data array, writing to its indices such that the array exists entirely in the L1 Cache.
6. Read the TSC Register and store its value.
7. Perform 26 Loads to data in the L1 Cache, with each Load followed by an MFENCE[19], ensuring the load completes before the next starts.
8. Read the TSC Register and store its value.
9. Calculate the latency of this load by getting the difference in the TSC reads, and subtracting the overhead of reading the timestamps to get the time for 26 Loads. Divide this value by 26 to get the Latency for a single Load.
10. In the latencies array, increment the appropriate index to indicate we have seen another L1 Load take the index's number of Cycles.

4.3.2 Key Features/Integral Components

Figure 4.5 shows a high-level overview of the algorithm, however, there are many subtleties that are important to note that cannot be seen from such a high-level.

1. **Timing Overhead**
   The overhead argument in `Line2` was calculated taking into account that there will be 26 `MFENCE` Flags/Instructions in the Critical Section, to ensure that the measured Latency doesn't account for the time the Pipeline may have been stalled waiting for Loads to become globally visible. This means that the overhead is calculated using the exact same algorithm as Figure 4.5 except the Critical Section contains only 26 `MFENCE` Flags/Instructions.
2. **Warm-Up Procedure**
   At the start of every iteration (i.e. `Line 7`) an omitted warmup() procedure is called. This refers to the discovery made in Section 4.1.3.2, that calling `CPUID`, `RDTSC`, and `RDTSCP` numerous times to essentially "warm them up", produces more repeatable and reliable results.
3. **Load Addresses**
   Between `Line 12` and `line 18` the algorithm states that 26 loads are made to consecutive indices of the L1 Data. However in the implementation, it is not consecutive indices, but the actual index is `N * STRIDE` where `STRIDE` is the number of 32-bit integers that would fit into a single line in the Cache. This is *extremely* important as it means that all loads hit different Cache Lines, which means that each load will definitely happen independently with the inclusion of the `MFENCE` Flags/Instructions.

4. **Timestamp Usage**
   Both start_timestamp and end_timestamp are used in Figure 4.6. This snippet shows that although the TSC is read, the two resultant 32-bit integers (for the upper and lower 32-bits of the 64-bit TSC value) are not converted to a 64-bit value. This is important because had the conversion been conducted within the Critical Section, it would have polluted the measured latencies and hence the result.

```
uint32_t start_hi, start_lo,end_hi, end_lo;
uint64_t start, end, latency;

start_timestamp(&start_hi, &start_lo);
/* Critical Section */
end_timestamp(&end_hi, &end_lo);

start   = ( ((uint64_t)start_hi << 32) | start_lo );
end     = ( ((uint64_t)end_hi   << 32) | end_lo   );
latency = (end - start);
```

Figure 4.6: Timestamp Functions Usage

5. **Boundary Checks**
   As the implementation is actually in c++, appropriate care was taken when performing tasks such as array indexing to ensure that abnormal latencies (that could result as a by-product of OS scheduling) do not cause segmentation faults or program crashes.

4.4 L2 Cache Latency

Building upon the conclusion of Section 4.1 and the Algorithm used in Section 4.3.1, this section will discuss in detail how measuring the latency of a Load to a Local L2 Cache is benchmarked. The same *Key Features/Components* mentioned in Section 4.3.2 still apply, and further important design decisions will be described.

4.4.1 Algorithm

4.4.1.1 Algorithm Pseudocode

```
# Returns the Latency of a single L2 Load
def getL2Latency(overhead):
    CorePin(0)                      # Pin Thread to Core 0
    latencies[500] = {0,0,...,0} # Init array of size 500 to all 0's
    for i in range(0,1000):
        data[L2_SIZE]               # Array that will fill L2 Cache

        # Fill the L1 & L2 Cache with target data.
        for x in range(0, L2_SIZE):
            data[x] = x

        # Fill the L1 Cache with subset of target data.
        for y in range(0, L1_SIZE):
            data[y] = y + 1

        start = start_timestamp()
        # Start Critical Section
        # Perform 26 Loads to data.
        load(data[L1_SIZE + 0])
        asm("MFENCE")
        load(data[L1_SIZE + 1])
        asm("MFENCE")
        ...
        load(data[L1_SIZE + 25])
        asm("MFENCE")
        # End Critical Section
        end   = end_timestamp()

        latency = (end - start - overhead) / 26
        latencies[latency]++      # Increment count of this latency seen

    printLatencies(latencies)      # Prints results
```

Figure 4.7: L2 Latency Algorithm

4.4.1.2 Algorithm Walkthrough

1. Pins the Process/Thread to a single Core
2. Declares an array (of size 500), where the i'th element is the number of times a Load was measured to have taken i Cycles.
3. Loops for 1000 times, to perform repeated trials.
4. Declare a data array that will be large enough to fill the L2 Cache.
5. Loop through the L2 Data array, writing to its indices such that the array exists entirely in the L1 and L2 Cache.
6. Loop through the first `N` indices of the L2 Data array, writing new values to those indices such that the first `N` elements exist in the L1 Cache, and the remaining `(L2_SIZE - N)` elements exist in the L2 Cache.
7. Read the TSC Register and store its value.
8. Perform 26 Loads to data in the L2 Cache, with each Load followed by an MFENCE[19], ensuring the load completes before the next starts.
9. Read the TSC Register and store its value.
10. Calculate the latency of this load by getting the difference in the TSC reads, and subtracting the overhead of reading the timestamps. Divide this value by 26 to get the Latency for a single Load.
11. In the latencies array, increment the appropriate index to indicate we have seen another L2 Load take the index's number of Cycles.

4.4.2 Key Features/Integral Components

Figure 4.7 shows a high-level overview of the algorithm for measuring L2 Load Latencies, all of the key features present in the L1 Latency Benchmark (Section 4.3.2) are also implemented within the L2 Latency Benchmark. There is also one more important design decision within the L2 Latency Benchmark:

- **Data Declaration**
  `Line 6` shows that the data to be used in the L2 Cache is newly declared within each trial. This is different to the L1 Latency algorithm in Section 4.3.1. This is to make sure that each trial has its own unique set of data. It is important for each trial to have its own set of data because we do not want data that exists from past trials to exist in the L1 Cache, which could possibly cause undesired L1 Hits when timing loads. It may be possible to flush all relevant Cache Lines to avoid this issue, but it is safer, easier and quicker to have each trial have its own data.

4.5 DRAM/MCDRAM Latency

The Benchmark that is used for determining DRAM Latency is also used for determining MCDRAM Latency. This is possible because we are able to use the `numactl`[1] terminal utility to choose which NUMA Node[2] to target for the program's execution. This means that MCDRAM or DRAM can be used as the next level of memory above the L2 Cache's, and we can write Benchmarks that measure loads beyond the L2 Cache, and depending on the run arguments, time the MCDRAM or DRAM.

4.5.1 Algorithm

4.5.1.1 Algorithm Pseudocode

```
# Returns the Latency of a single DRAM Load
def getDRAMLatency(overhead):
    CorePin(0)                  # Pin Thread to Core 0
    latencies[500] = {0,0,...,0} # Init array of size 500 to all 0's
    data[1000]              # Array sufficiently sized.
    for i in range(0,1000):
        # Ensure this data does not exist in any Caches.
        for x in range(0, 1000):
            CLFLUSH(x)          # Flush this line from all Caches.

        start = start_timestamp()
        # Start Critical Section
        # Perform 26 Loads to data.
        load(data[0])
        asm("MFENCE")
        load(data[1])
        asm("MFENCE")
        ...
        load(data[25])
        asm("MFENCE")
        # End Critical Section
        end   = end_timestamp()

        latency = (end - start - overhead) / 26
        latencies[latency]++      # Increment count of this latency seen

    printLatencies(latencies)     # Prints results
```

Figure 4.8: DRAM Latency Algorithm

[2]Dependent on L2 Cache and MCDRAM config, MCDRAM and DRAM will be exposed as separate NUMA Memory Regions that can be targeted by `numactl`

4.5.1.2 Algorithm Walkthrough

1. Pins the Process/Thread to a single Core Declares an array (of size 500), where the i'th element is the number of times a Load was measured to have taken i Cycles.
2. Declare a data array that will be large enough to perform 13 Loads to.
3. Loops for 1000 times, to perform repeated trials.
4. Loop through the data array, flushing the elements from the Cache's, invalidating the lines, and writing back values to DRAM memory.
5. Read the TSC Register and store its value.
6. Perform 13 Loads to data that has been flushed from the Caches, with each Load followed by an MFENCE[19], ensuring the load completes before the next starts.
7. Read the TSC Register and store its value.
8. Calculate the latency of this load by getting the difference in the TSC reads, and subtracting the overhead of reading the timestamps. Divide this value by 26 to get the Latency for a single Load.
9. In the latencies array, increment the appropriate index to indicate we have seen another DRAM Load take the index's number of Cycles.

4.5.2 Key Features/Integral Components

Figure 4.8 shows a high-level overview of the for measuring [MC]DRAM Load Latencies, all of the key features present in the L1 Latency Benchmark (Section 4.3.2) are also implemented within the L2 Latency Benchmark. There are also two more important design decision within the L2 Latency Benchmark:

- **Load Ordering**
  Although the [MC]DRAM Load Latency Benchmark performs reads to different lines in memory, the reads must be carefully chosen/ordered in order to prevent data from being pre-fetched. This is important, because if we were to load from line 0, then line 1, then line 2, then subsequent lines will be pulled from memory (as part of a pre-fetching strategy) to the L2 Cache, defeating the purpose of timing loads to [MC]DRAM. In the Benchmark implementation the lines are loaded in the following order: [12,0,11,1,10,2,9,3,8,4,7,5,6] which prevented the pre-fetcher from engaging.

4.6 Remote Cache Latency(Coherence Miss Latencies)

In order to measure remote Cache access latencies, we must have multiple threads running concurrently that are tightly synchronised, each with their own task.

4.6.1 Thread Structure

The benchmark program begins by creating a thread for each core, and in the creation of the thread, passes the Core that the Thread should be pinned to, the task the thread should perform, and the state in which the Thread should perform its task. Therefore the following flow chart shows the behaviour of all threads within the program:

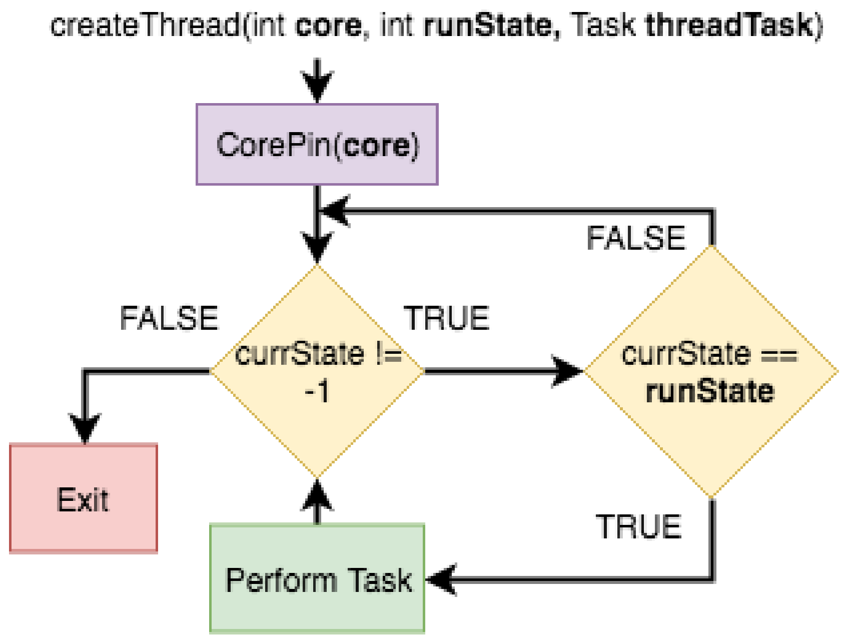

createThread(int **core**, int **runState**, Task **threadTask**)
CorePin(**core**)
FALSE
FALSE
TRUE
currState != -1
currState == **runState**
Exit
TRUE
Perform Task

Figure 4.9: Thread Flow Chart

The flowchart visualises what is essentially a while loop that runs while we are still progressing through the benchmark, and an if statement that will trigger the desired behaviour under certain global state conditions.

The benchmarks are capable of measuring Remote L1 Load Latencies in the Modified and Exclusive states from the `MESIF` Cache Coherence Protocol(which was briefly discussed in Section 2.2.2 and Section 2.2.3) but were not fully completed for loads to lines in the Shared state. It is important to note that it was chosen that only 1 thread is pinned to a Core, as this will avoid the complications of assigning multiple threads to the same Core that were briefly outlined in Section 3.2. It was observed that filling every logical core[3] would increase the execution times of benchmarks, and produce incorrect results, likely as a result of the threads sharing Core resources.

[3]Logical Core: If a Core is capable of executing 4 Threads simultaneously, the OS sees this Core as 4 Logical Cores.

4.6.2 Thread Tasks

Tasks are pinned to cores depending on the latency we want to measure. We will refer to the *Base Core* as the Core where the Cache Line's of interest exists in the desired State, and the *Target Core* to be the Core we wish to access said lines in the *Base Core* from. For example, to measure the latency of Core 5 accessing data that exists in Core 0's L1 Cache in Modified State, the *Base Core* would be Core 0, and the *Target Core* would be Core 5. In the case of measuring latencies to data in the Shared state, we refer to the *Alt Core* as the Core which will also hold the shared data alongside the *Base Core*.

4.6.2.1 Write Data

Figure 4.10 shows how a thread could own Modified State data. It involves iterating over the shared data, and assigning new values to each index. Upon completing the for loop, all the data will exist in the Modified state in the L1 Cache of the Core that the thread was pinned to.

```
for (int i=0; i < 300; i++) {
    shared_data[i] = i;
}
currState++;
```

Figure 4.10: Write Data Task - C++ Code

4.6.2.2 Read Data

Figure 4.11 shows how a thread could own Exclusive/Shared data. It involves iterating over the shared data, and performing loads to the data (using the Load technique discussed in Section 4.1.4). Upon completing the for loop, all the data will exist in either Exclusive or Shared state in the L1 Cache of the Core that the thread was pinned to, depending on what tasks the other threads are assigned.

```
for (int i=0; i < 300; i++) {
    asm volatile (
        "\n\tmov %1, %0"
        "\n\tMFENCE"
        :"=r"(shared_data[i])
        :"r"(shared_data[i]):
    );
}
currState++;
```

Figure 4.11: Read Data Task - C++ Code

4.6.2.3 Time Data Access

The thread purposed with timing the remote access uses the same method of accessing the data that was used in the previous latency benchmarks, in Section 4.1, which results in the same code used in Figure 4.6 to time 26 Loads to different Cache Lines within the range of `shared_data[0]` to `shared_data[300]`. Figure 4.12 again shows the code snippet. The latencies measured are again stored in an array, where index `i` of the array contains the number of times a latency of `i` cycles was measured.

```
start = start_timestamp()
# Start Critical Section
# Perform 26 Loads to shared data.
load(shared_data[0])
asm("MFENCE")
load(shared_data[1])
asm("MFENCE")
...
load(shared_data[25])
asm("MFENCE")
# End Critical Section
end   = end_timestamp()

latency = (end - start - overhead) / 26
latencies[latency]++     # Increment count of this latency seen
```

Figure 4.12: Pseudocode for Timing Data Accesses

4.6.2.4 Dead Task

For cores that are not in use, there is a ”Dead Task” which simply stalls until the `currState` is set to -1 (marking the experiment is over), then the thread returns/exits. These dead tasks are allocated to cores to ensure that each core has an active running thread, to ensure that all Core’s are active.

4.6.3 Remote L1 Modified Read

4.6.3.1 Thread/Task Allocation

To perform a load to a Modified Cache Line in a Remote L1 Cache, we must have our Base Core modify the data, and then our Target Core should perform the timed accesses to the data that is in the Modified State. Using the Flow Chart in Figure 4.9 as our `createThread()` method, we can use the following pseudocode to generate our benchmark:

```
1  currState = 0
2  // Set Write Task on State 1
3  createThread(BASE_CORE, 1, write)
4  // Set Timing Task on State 2
5  createThread(TARGET_CORE, 2, time_access)
6  /* Assign Dead Tasks to all other Cores */
7  for (int i=0; i < NUM_CORES; i++) {
8      if (i != BASE_CORE && i != TARGET_CORE)
9          createThread(i, -1, dead_task)
10 }
11 /* Set State to 1 to get Tasks Started */
12 currState = 1
13 while (currState != 3) { /* Spin */ }
14 /*Set State to -1 to end all Threads/Tasks */
15 currState = -1
```

Figure 4.13: Remote Modified Read - Thread/Task Allocation Pseudocode

4.6.3.2 State Diagram

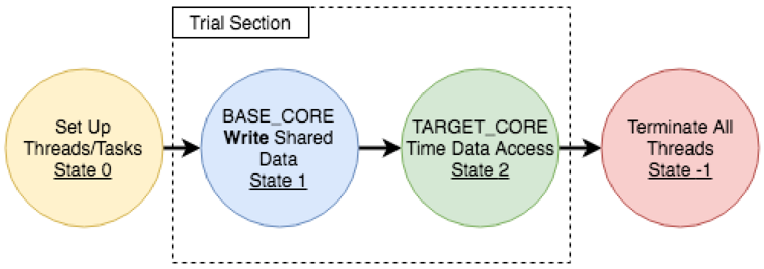

Trial Section
Set Up Threads/Tasks State 0
BASE_CORE Write Shared Data State 1
TARGET_CORE Time Data Access State 2
Terminate All Threads State -1

Figure 4.14: Remote L1 Modified Read State Machine

4.6.3.3 Summary

In summary, to obtain a latency measurement for a load to a Modified Line in a remote L1 Cache, there must be a Core which performs **writes** to a set of shared data, and in turn, sets it to Modified state in the Cores local Cache. Then, another Core must attempt to read the same set of shared data, which will incur a local Cache miss, and the data will be fetched from the *Base Core* which owns the data in Modified State. The key features outlined in Section 4.3.2, 4.4.2, and 4.5.2 all apply including the *Data Declaration Consideration* outlined in Section 4.4.2, which has minor modifications in implementation to ensure the same global and shared variable can be used independently and uniquely within the *Trial Section* outlined in Figure 4.14. The repeated trials are performed by the following code snippet as a drop-in replacement for Lines 12 to 15 in Figure 4.13:

```
for (int i = 0; i < 1000; i++) {    // Do 1000 Trials.
    currTask = 0;
    // Re-allocate shared data to invalidate across all Cores.
    shared_data = (int*)malloc(L2_SIZE_B);
    currTask = 1;

    /* Threads Complete their Tasks */

    while (currTask != 3) { /* Wait for all 2 Tasks to complete.

    // Delete our shared data to prevent memory leaks.
    delete shared_data;
}
currTask = -1;  // Signal threads to terminate.
```

Figure 4.15: Remote Modified Read - Repeated Trials

4.6.4 Remote L1 Exclusive Read

4.6.4.1 Thread/Task Allocation

To perform a load to a Cache Line in the Exclusive state in a Remote L1 Cache, we must have our Base Core load the data, and then our Target Core should perform the timed accesses to the data in the Exclusive State. Again, using the Flow Chart in Figure 4.9 as our `createThread()` method, we can use the following pseudocode to generate our benchmark:

```
currState = 0

// Set Read Task on State 1 on Core BASE_CORE
createThread(BASE_CORE, 1, read)
// Set Timing Task on State 2 on Core TARGET_CORE
createThread(TARGET_CORE, 2, time_access)
/* Assign Dead Tasks to all other Cores */
for (int i=0; i < NUM_CORES; i++) {
    if (i != BASE_CORE && i != TARGET_CORE)
        createThread(i, -1, dead_task)
}
/* Set State to 1 to get Tasks Started */
currState = 1
while (currState != 3) { /* Spin */ }
/*Set State to -1 to end all Threads/Tasks */
currState = -1
```

Figure 4.16: Remote Exclusive Read - Thread/Task Allocation Pseudocode

4.6.4.2 State Diagram

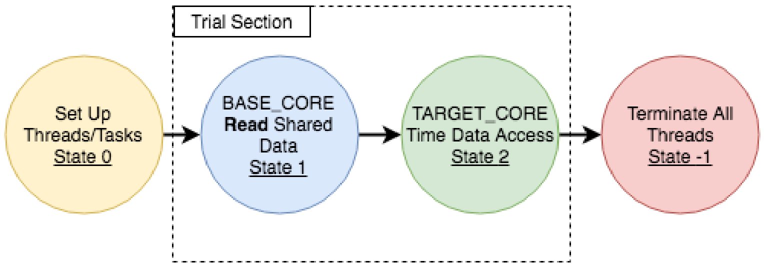

Trial Section
Set Up Threads/Tasks State 0
BASE_CORE Read Shared Data State 1
TARGET_CORE Time Data Access State 2
Terminate All Threads State -1

Figure 4.17: Remote L1 Exclusive Read State Machine

4.6.4.3 Summary

In summary, to obtain a latency measurement for a load to an Exclusive Line in a remote L1 Cache, there must be a Core which performs **loads** to a set of shared data, and in turn, sets it to Exclusive state in the Cores local Cache. Then, another Core must attempt to read the same set of shared data, which will incur a local Cache miss, and the data will be fetched from the *Base Core* which owns the data in the Exclusive State. The key features outlined in Section 4.3.2, 4.4.2, and 4.5.2 all apply including the *Data Declaration Consideration* outlined in Section 4.4.2, which has minor modifications in implementation to ensure the same global and shared variable can be used independently within the *Trial Section* outlined in Figure 4.14. The repeated trials are performed by the following code snippet as a drop-in replacement for Lines 13 to 16 in Figure 4.16:

```
// Do 1000 Trials.
for (int i = 0; i < 1000; i++) {
    currTask = 0;
    // Re-allocate shared data to invalidate across all Cores.
    shared_data = (int*)malloc(L2_SIZE_B);
    currTask = 1;

    /* Threads Complete their Tasks */

    while (currTask != 3) {
        /* Wait for all 2 Tasks to complete. */
    }

    // Delete our shared data to prevent memory leaks.
    delete shared_data;
}
currTask = -1;  // Signal threads to terminate.
```

Figure 4.18: Remote Exclusive Read - Repeated Trials

4.6.5 Remote L1 Shared Read

4.6.5.1 Thread/Task Allocation

To perform a Remote L1 Read to Shared Cache Lines, we must have our *Base Core* read the shared data, then have another Core (our *Alt Core*) read the same shared data, and then finally our Target Core should perform the timed accesses to the shared lines in the Shared State. This scenario is a little more complex than the previous two, as it involves the use of a third Core, which makes the benchmark more complex. It is more complex because we cannot definitively say if it will be the *Base Core* or the *Alt Core* that will own the lines in the *Forward State/Shared State*, meaning that we must take care when deciding which Cores to use. Again, using the Flow Chart in Figure 4.9 as our `createThread()` method, we can use the following pseudocode to generate our Benchmark:

```
currState = 0
// Set Read Task on State 1
createThread(BASE_CORE, 1, read)
// Set Read Task on State 2
createThread(ALT_CORE, 2, read)
// Set Timing Task on State 3
createThread(TARGET_CORE, 3, time_access)
/* Assign Dead Tasks to all other Cores */
for (int i=0; i < NUM_CORES; i++) {
    if (i != BASE_CORE && i != TARGET_CORE && i != ALT_CORE)
        createThread(i, -1, dead_task)
}
/* Set State to 1 to get Tasks Started */
currState = 1
while (currState != 4) { /* Spin */ }
/*Set State to -1 to end all Threads/Tasks */
currState = -1
```

Figure 4.19: Remote Shared Read - Thread/Task Allocation Pseudocode

4.6.5.2 State Diagram

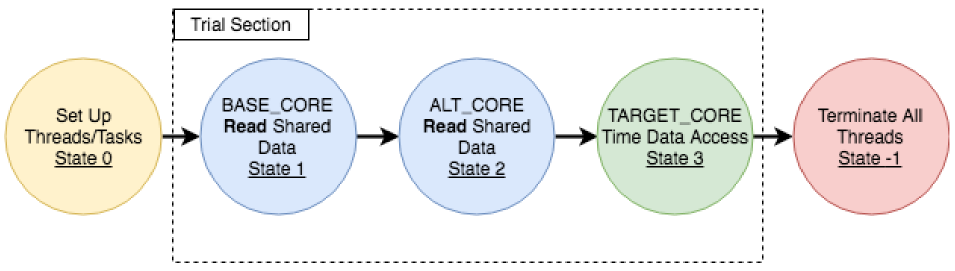

Trial Section
Set Up Threads/Tasks State 0
BASE_CORE Read Shared Data State 1
ALT_CORE Read Shared Data State 2
TARGET_CORE Time Data Access State 3
Terminate All Threads State -1

Figure 4.20: Remote L1 Shared Read State Machine

4.6.5.3 Summary

In summary, to obtain a latency measurement for a load to a Shared Line in a remote L1 Cache, there must be a Core which performs **loads** to a set of shared data, and in turn, sets it to Exclusive state in that Cores local Cache. Then, another Core must perform the same set of **loads**, which will miss in the Core's Local Cache, and the Cache Lines will be retrieved from a Remote Core (the *Base Core*), and the State of those Cache Lines will be updated to Shared State in both Cores. Finally, another Core must attempt to load the same set of shared data, which will incur a local Cache miss, and the data will be fetched from the Core that owns the Cache Lines in the *Forward State*. As with the Modified/Exclusive benchmarks, the key features outlined in Section 4.3.2, 4.4.2, and 4.5.2 all still apply, including the Data Declaration Consideration outlined in Section 4.4.2. The repeated trials are performed by the following code snippet:

```
// Do 1000 Trials.
for (int i = 0; i < 1000; i++) {
    currTask = 0;
    // Re-allocate shared data to invalidate across all Cores.
    shared_data = (int*)malloc(L2_SIZE_B);
    currTask = 1;

    /* Threads Complete their Tasks */

    while (currTask != 4) {
        /* Wait for all 3 Tasks to complete. */
    }

    // Delete our shared data to prevent memory leaks.
    delete shared_data;
}
currTask = -1;  // Signal threads to terminate.
```

Figure 4.21: Remote Shared Read - Repeated Trials

4.7 DRAM/MCDRAM Bandwidth

4.7.1 STREAM Benchmark

The STREAM Benchmark is designed for evaluating High Performance Computing (HPC) systems. HPC systems typically work with large data sets that cannot fit within a typical Cache system. Data of this nature means that processors are typically spending a lot of time waiting for data to be transferred to-and-from memory systems such as DRAM or MCDRAM, and the STREAM benchmark evaluates how much data chips such as the KNL can process per second. In order to evaluate the performance of a chip in an HPC context, the STREAM benchmark tests four ”kernels”, which are essentially memory and CPU intensive operations that are common in HPC applications.

Table 4.3: STREAM Kernels

Kernel Name	Kernel Operation(s)
COPY	`a(i) = b(i)`
SCALE	`a(i) = q*b(i)`
SUM	`a(i) = b(i) + c(i)`
TRIAD	`a(i) = b(i) + q*c(i)`

Table 4.3 shows the different kernels operations. As we can see, the `COPY` Kernel copies data from one memory location to another. The `SCALE` kernel scales up all data in a region by a certain coefficient. The `SUM` kernel sums data from two sources and stores in a third. Finally, the `TRIAD` performs an amalgamation of all the previous kernels.

4.7.2 STREAM Usage

It is important to ensure that when performing STREAM benchmarks, we perform the benchmarks using the guidelines provided.

4.7.2.1 Compilation

Compilation of the STREAM benchmark is important in ensuring that the benchmark can be run in a parallel fashion properly. As such the code was compiled using the following command:

```
cc stream.c -O3 -o stream -fopenmp -DSTREAM_ARRAY_SIZE=64000000
```

It is important to include the `-fopenmp` flag, as this will ensure that the code that is marked with `"#pragma omp parallel for"` will be parallelised appropriately. We also define the `STREAM_ARRAY_SIZE` to be sufficiently large in accordance with the guidelines.

4.7.2.2 Running

In order to run the STREAM benchmark in a reliable parallel fashion, it must be invoked by a run-script that exports appropriate environment variables.

```
1 export OMP_PROC_BIND=true
2 OMP_NUM_THREADS=1   numactl --membind 0 ./stream
```

Figure 4.22: Sample of STREAM Run Script

Figure 4.22 shows a sample of the `run.sh` script used to invoke the STREAM benchmark. Line 1 exports an environment variable which will ensure that OpenMP Threads cannot be migrated between Cores. Then "`OMP_NUM_THREADS=1`" specifies that there should be exactly 1 OpenMP Thread created. The full run script rials with different numbers of OpenMP Threads, scaling exponentially from 1 Thread to 128 Threads. The "`numactl --membind 0`" call ensures that the program is run on the appropriate NUMA region. Although the memory location here is bound to NUMA Region 0, this value is substituted with the parameter of the run script, such that different NUMA Regions can be tested.

Chapter 5

Results

Results were obtained on a 64 Core Intel Xeon Phi CPU 7210 Clocked at 1.30GHz.

5.1 Latencies

5.1.1 Intra-Tile Latencies

Intra-Tile Latencies are Latencies of accessing data in Modified/Exclusive State. In the context of these results, **Local** refers to the same Core, and **Remote** refers to the other *Core* that is on the same *Tile*.

Table 5.1: Intra-Tile Latencies

Cache	Latency
Local L1 Cache	0.77ns (1 Cycle)
Local L2 Cache	5.38ns (7 Cycles)
Remote L1 Cache[M/E]	11.54ns (15 Cycles)

5.1.2 Inter-Tile Latencies

Inter-Tile Latencies are the Latencies of accessing data in other Tile's L1 Cache in Modified/Exclusive State. In the context of these results, **Local** refers to the same *NUMA Region*, and **Remote** refers to another *NUMA Region*.

Table 5.2: Inter-Tile Latencies

L2 Cache Config	Latency
All-to-All	26.15ns (34 Cycle)
Quadrant	26.15ns (34 Cycle)
Sub NUMA Local	25.38ns (33 Cycle)
Sub NUMA Remote	27.69ns (36 Cycle)

5.1.3 MCDRAM/DRAM Latencies

MCDRAM/DRAM Latencies are the Latencies of accessing data that exists only in DRAM or MCDRAM. In the context of these results, **Local** refers to the memory in the same *NUMA Region*, and **Remote** refers to memory in another *NUMA Region*.

Table 5.3: MCDRAM Latencies

L2 Cache Config	Memory Config	Latency
All-to-All	Flat	169.23ns (220 Cycles)
All-to-All	Cache	169.23ns (220 Cycles)
Quadrant	Flat	165.38ns (215 Cycles)
Quadrant	Cache	165.38ns (215 Cycles)
Sub NUMA Local	Flat	161.54ns (210 Cycles)
Sub NUMA Local	Cache	161.54ns (210 Cycles)
Sub NUMA Remote	Flat	173.85ns (226 Cycles)
Sub NUMA Remote	Cache	173.85ns (226 Cycles)

Table 5.4: DRAM Latencies

L2 Cache Config	Memory Config	Latency
All-to-All	Flat	134.62ns (175 Cycles)
Quadrant	Flat	133.08ns (173 Cycles)
Sub NUMA Local	Flat	127.69ns (166 Cycles)
Sub NUMA Remote	Flat	138.46ns (180 Cycles)

5.2 Bandwidths

5.2.1 All-to-All

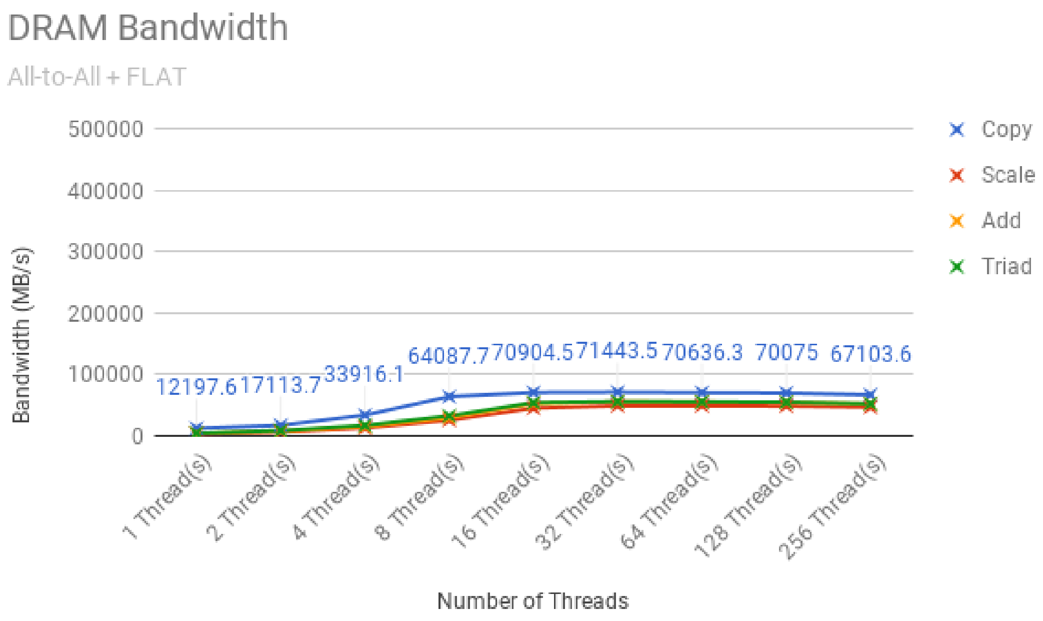


Figure 5.1: DRAM - Flat Mode

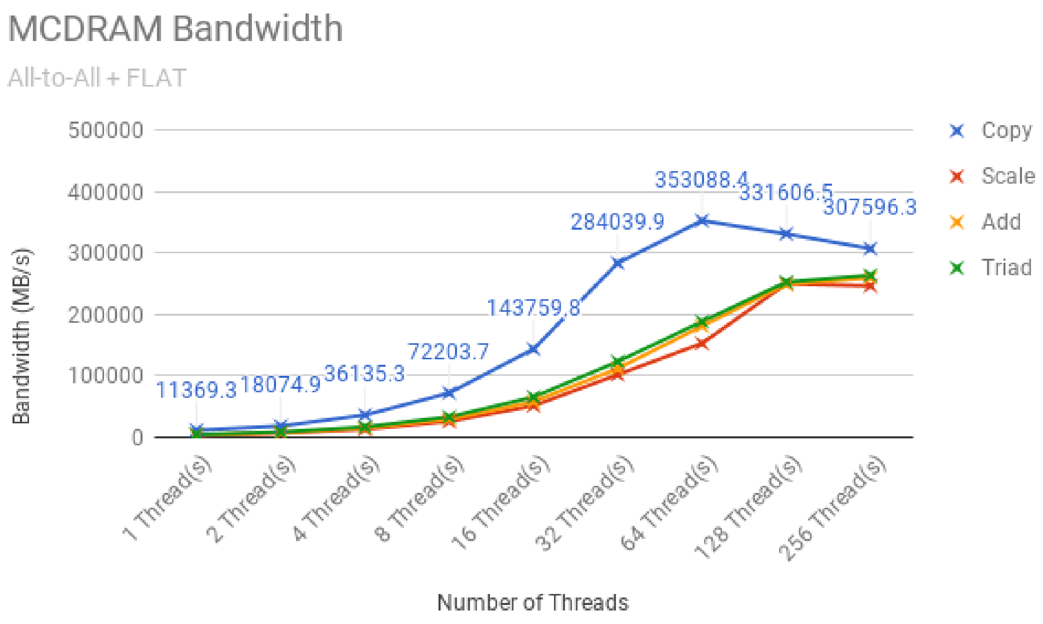


Figure 5.2: MCDRAM - Flat Mode

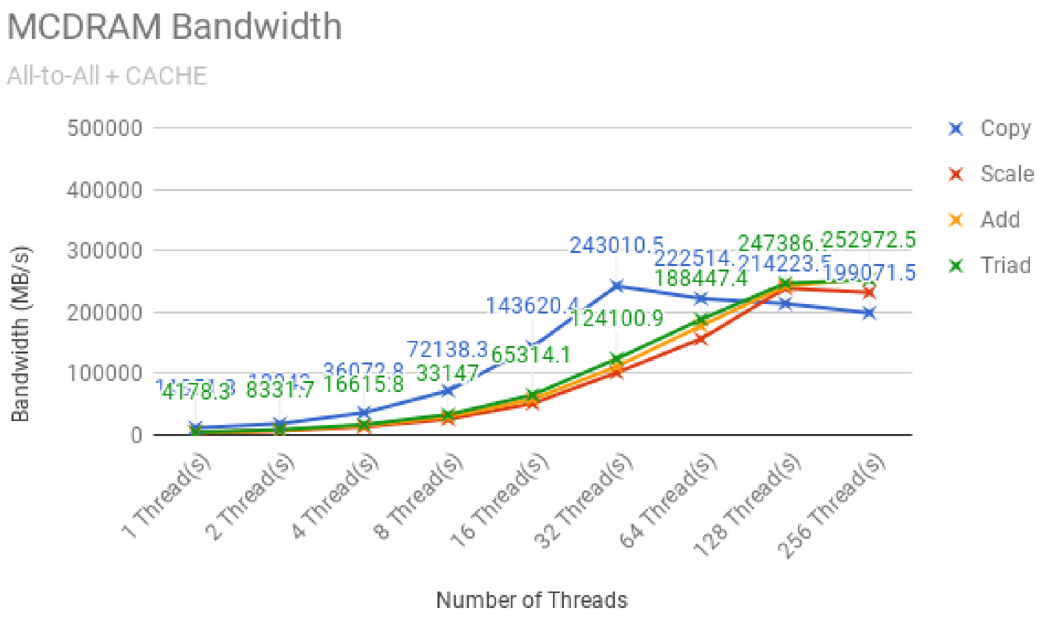


Figure 5.3: MCDRAM - Cache Mode

5.2.2 Quadrant

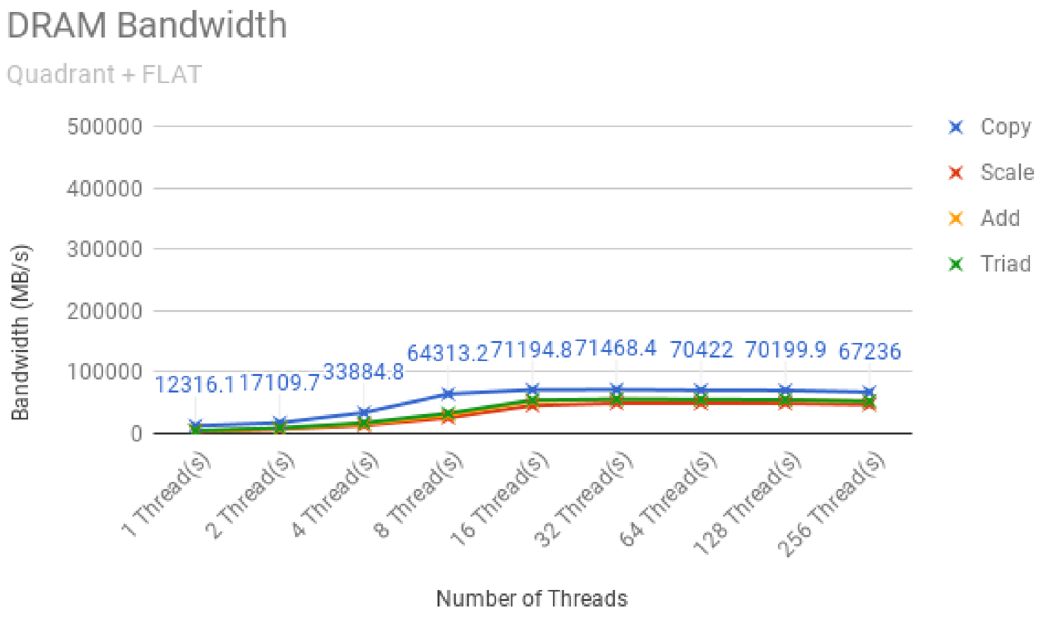


Figure 5.4: DRAM - Flat Mode

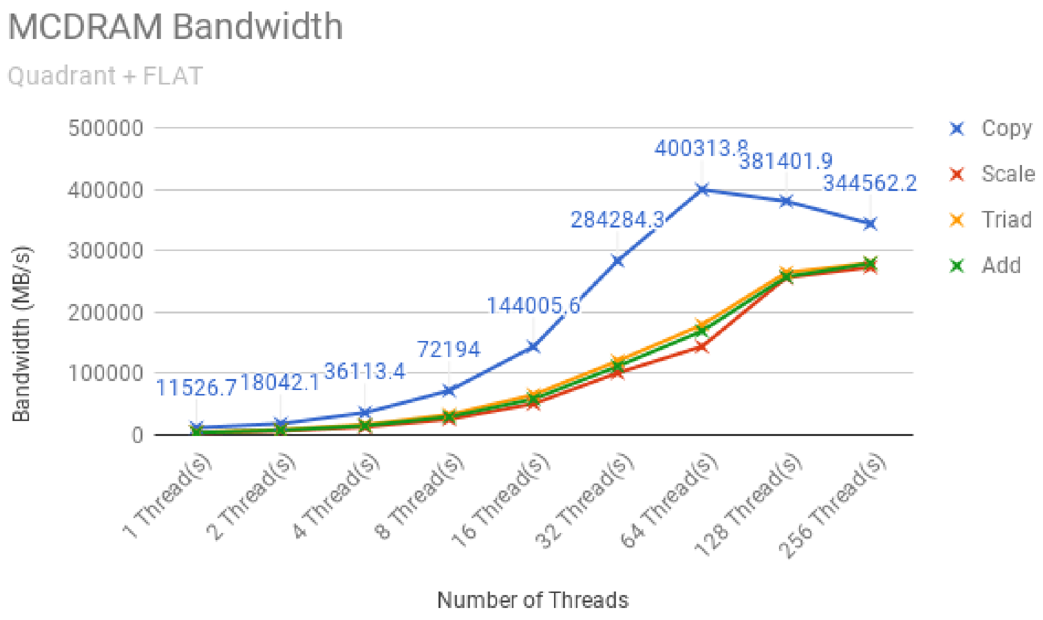


Figure 5.5: MCDRAM - Flat Mode

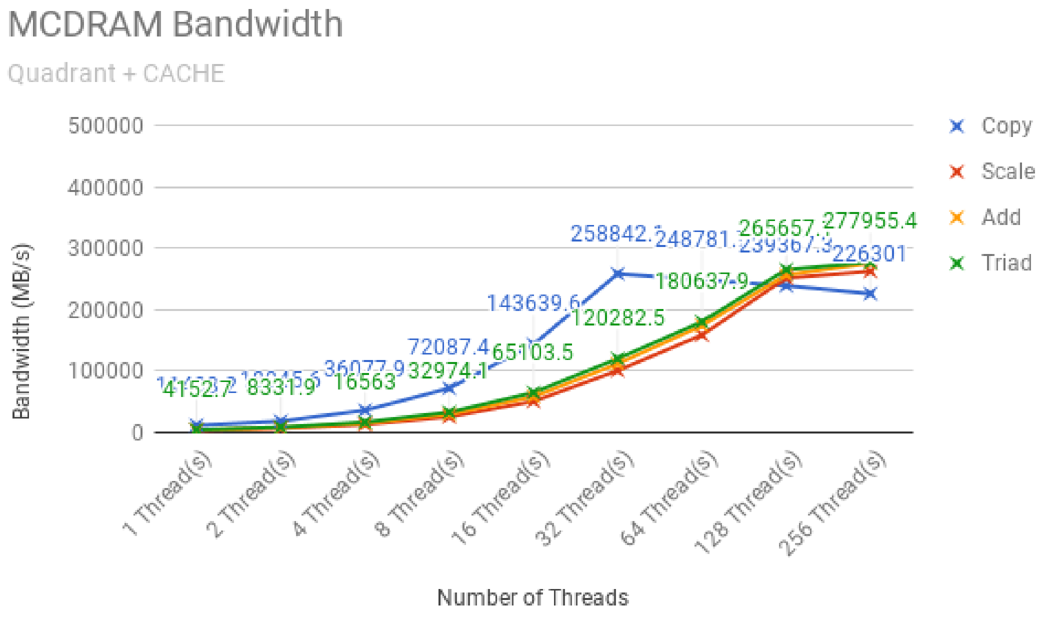


Figure 5.6: MCDRAM - Cache Mode

5.2.3 Sub-NUMA

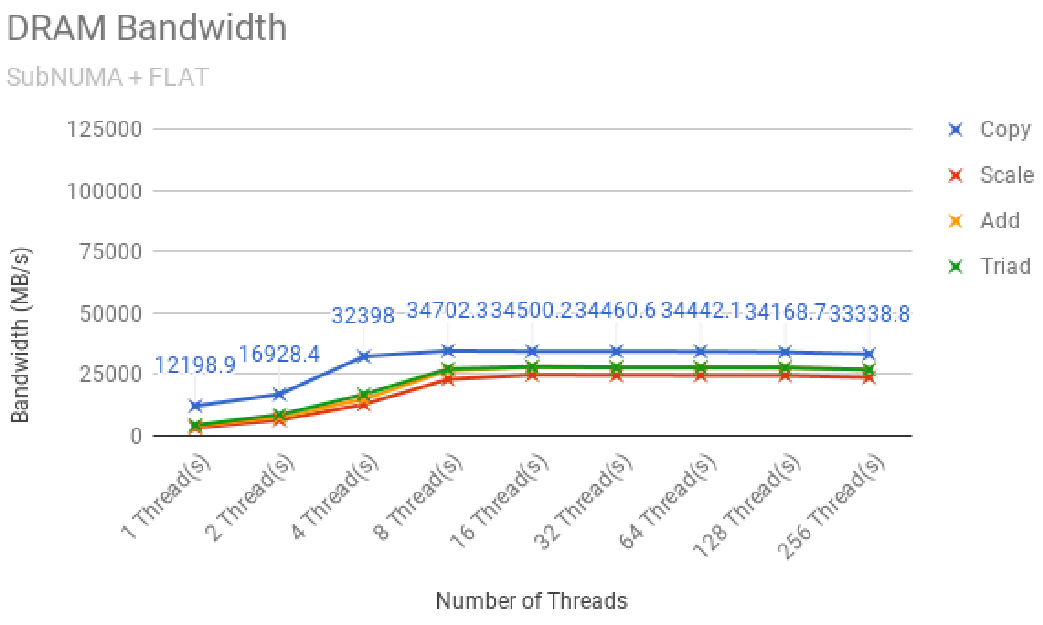


Figure 5.7: DRAM - Flat Mode

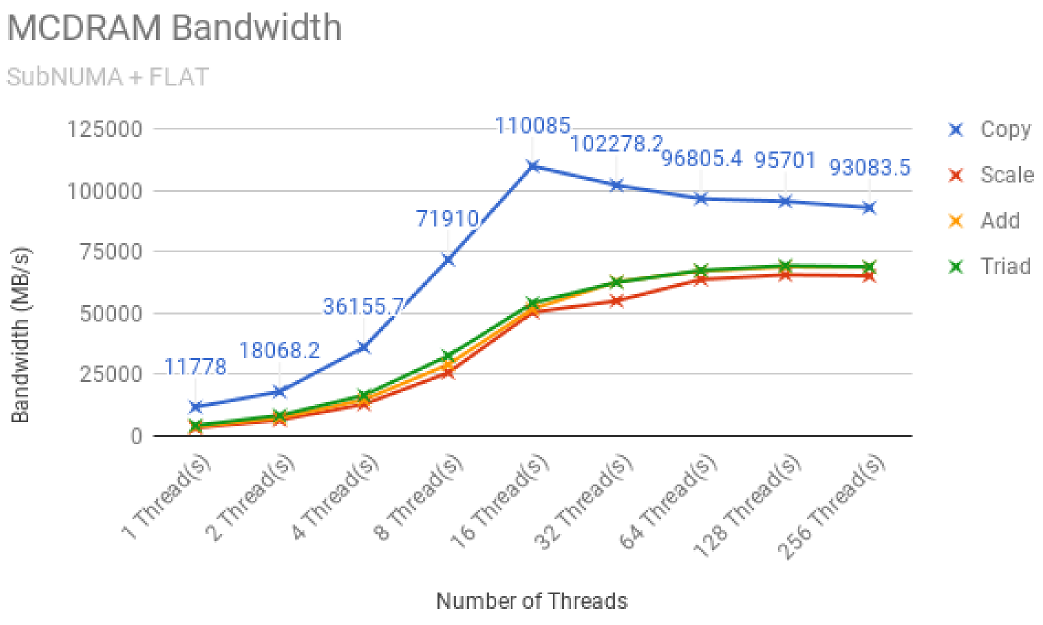


Figure 5.8: MCDRAM - Flat Mode

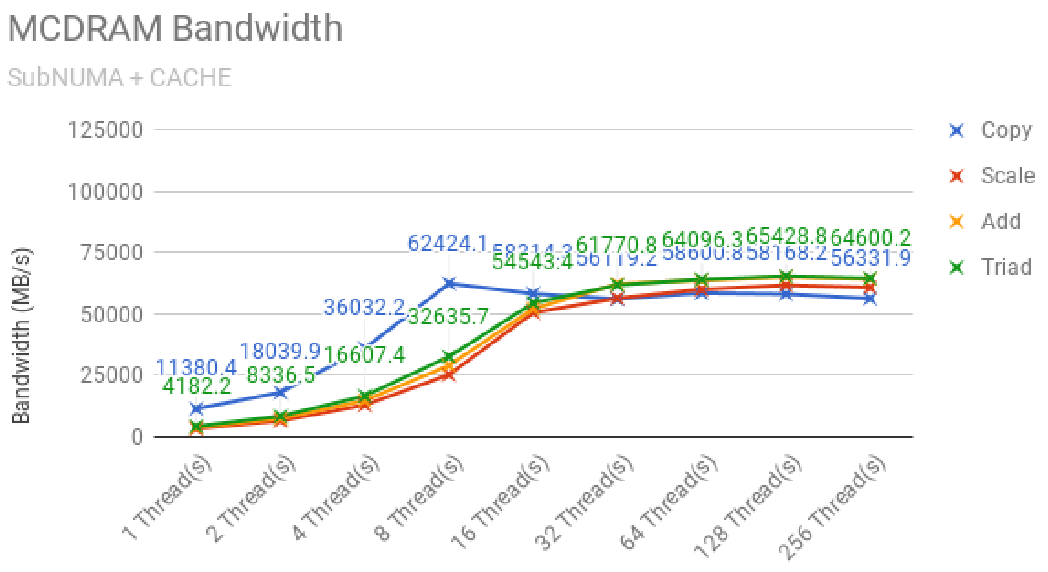


Figure 5.9: MCDRAM - Cache Mode

Chapter 6

Discussion

When discussing results, I will refer to the system presented in the paper by Hackenberg, Molka and Nagel[5] as the ”Nehalem” system.

6.1 Latency Conclusions

6.1.1 Intra-Tile Results

6.1.1.1 Local L1 Cache

The measured Latencies in Table 5.1 follow an expected trend. We see that an L2 Access takes longer than an L1 Access, and that Accessing data in the Tile's adjacent Core L1 takes roughly twice the time of an L2 Access. However, it is possible that the results could be incorrect. If we refer to the Nehalem system, we see that their test system is a 2.6GHz Chip with a measured L1 Latency of 1.3ns/4 Cycles. Our benchmarks were run on a 1.3GHz chip and resulted in 0.77ns/1 Cycle. It is unlikely that Intel (roughly) halved the latency of an L1 Cache access, and is it more likely that there is some error within the timing mechanism that could cause incorrect rounding of Cycles when measuring multiple loads. If we consider an Intel article published by Eric G.[8], it specifies that the L1 Data Cache has two 64B Load Ports. Despite including Memory Fences after every load, it could be possible that the underlying architecture is able to process 2 Loads simultaneously. If it were the case that our microbenchmark loads were processed two at a time, then our measured latency would, in fact, be roughly double the observed. This would make our L1 Latency be 1.54ns, which aligns more with observed Latencies on other Intel architectures. Had more time been available, more work could have been done to deeper investigate how the number of loads timed would affect the measured latency.

6.1.1.2 Local L2 Cache

The Local L2 Cache Latency measured (Table 5.1) was 5.38ns/7 Cycles. This seems like a reasonable value for an L2 Latency, and when comparing to the Nehalem system, we see that that the Nehalem system had a measured Local L2 Cache Latency is 3.4ns/10 Cycles. Given that the Nehalem system's L2 Cache is 256KB, and our test system's L2 Cache is 1024KB, it would be feasible for the Latency of accessing our system's larger L2 Cache to be higher. Similar to the L1 Cache, it could be possible for the L2 Cache to be dual-ported, and therefore could be processing two loads simultaneously. More work could be done to verify the integrity of the results for example by performing benchmarks on a range of loads. Error may also have been introduced due to the methodology for generating L1 Misses/L2 Hits, perhaps investigating methodologies of properly thrashing[1] the L1 Cache with dead data would have provided more reliable results.

6.1.1.3 Remote L1 Cache

The measured Latency for a Remote L1 Cache on a Tile was 11.54ns/15 Cycles.

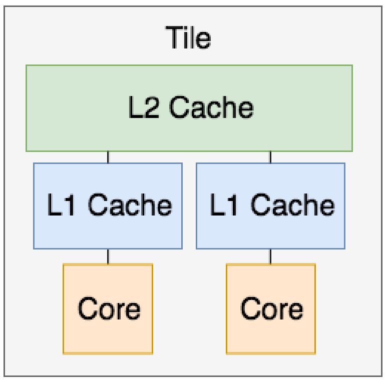


Figure 6.1: Tile Memory Pictorial

If we consider Figure 6.1 to be the architecture of the memory system within a Tile, it would make sense that our measured Latency would be 15 Cycles. If we consider that it would take 7 Cycles to determine if the Line exists in the L2, it would then make sense for another 7 Cycles to retrieve data from the remote L1 Cache, and deliver it to the consuming Core. However as we have already discussed the possible issues that could prevent accurate measurements, we must also consider that similar issues could be carried over into or made more complex in this microbenchmark.

[1]Thrashing the L1 Cache: Putting the L1 Cache under intense usage, in order to thoroughly remove all existing entries.

6.1.2 Inter-Tile Results

Unfortunately, results for accessing data in the Shared state were not reliably measured due to inconsistent and bizarre measured latencies. This may have been due to a lack of understanding of how the MESIF Protocol works, and lack of a deterministic way of determining which Core would be assigned the *Forward* state. When considering the Modified/Exclusive Latencies, in the respective L2 Modes, measured Modified and Exclusive latencies were identical, and are shown in Table 5.2 as a single Latency. There definitely exists an error in the results, as we would expect the latency of accessing remote data in Quadrant mode would be lower than the latency of accessing remote data in All-to-All as per Intel's Presentation on KNL[17]. However, an encouraging trend shows that accessing remote data in a Local Sub-NUMA configuration is faster than all other Cache configurations. It should also be expected that accessing data in a remote NUMA region would incur a large latency, as in theory, it should involve an access to main memory in the fashion detailed in Section 3.1.2(Under *Sub-NUMA Clustering*). Given that the measured latency for accessing data in a remote NUMA region was 27.69ns/36 Cycles, it is safe to say that an error exists in the experiment/methodology. When conducting the experiments, I ran the programs using the `numactl` program, ensuring that the NUMA region the program was loaded into was chosen appropriately for measuring remote accesses, verifying that tasks were pinned to appropriate Core using the output of calling `numactl -H`[2]. Pinning Tasks to Cores on the same Tile does drastically reduce the Latency(see "Remote L1 Cache" in Table 5.1) which indicates that the results are somewhat correct, however, I suspect that perhaps thread placement was not performed properly with respect to the NUMA regions, and had more time been available, research into how to validate this could have highlighted potential issues or verified existing results. In conclusion, I do not believe the results to be reliable, but the consistency of the results and the general trends indicate that there is a solid foundation for microbenchmarking cache coherence latencies.

6.1.3 MCDRAM/DRAM Latencies Results

When conducting MCDRAM/DRAM Latencies, care was taken to ensure that the Latencies measured were accesses to data on a single NUMA Node, to avoid the Latencies of cross-NUMA accesses.

John McCalpin has conducted microbenchmarks to measure the latencies of the MCDRAM and DRAM in different L2 Cache/Memory Configurations[13]. This is useful as it allows us to compare the results obtained for correctness, which can, in turn, validate aspects of our benchmark suite.

As can be seen in Table 5.3, the Memory Configuration has no effect on Latency. The following relationship on access latencies holds for both MCDRAM and DRAM:

$$\text{Sub-NUMA Local} \leq \text{Quadrant} \leq \text{All-to-All} \leq \text{Sub-NUMA Remote} \tag{6.1}$$

[2]`numactl -H`: Lists all NUMA regions in the system, and which cores exist in which regions.

Equation 6.1 shows an observed trend that we would expect to see in our results. Furthermore, we can compare our results to those obtained by John McCalpin, and we see the following similarities/differences:

Table 6.1: **DRAM** Latencies Verification (Flat Mode)

L2 Cache Config	McCalpin's Average Latency	Our Average Latency
All-to-All	131.8ns	134.62ns
Quadrant	130.4ns	133.08ns
Sub NUMA Local	128.2ns	127.69ns
Sub NUMA Remote	133.1ns	138.46ns

Table 6.2: **MCDRAM** Latencies Verification (Flat Mode)

L2 Cache Config	McCalpin's Average Latency	Our Average Latency
All-to-All	155.9ns	169.23ns
Quadrant	154.0ns	165.38ns
Sub NUMA Local	150.5ns	161.54ns
Sub NUMA Remote	156.8ns	173.85ns

Table 6.1 shows a comparison of our measured DRAM Latencies with John McCalpin's. The comparison shows that although our results are not identical (which is to be expected when dealing with DRAM), we follow a close trend. This trend can be seen if we look at the latency difference between different L2 Cache Configs; for example, the difference in latency between All-to-All and Quadrant was roughly 1.4ns in our measurements, and McCalpins. Similar observations can be made when comparing other modes, but with larger margins of error. This property again indicates that our microbenchmarks follow a correct design, but errors exist. I would hypothesise that some error exists within the timing mechanism, due to this trend of slights inconsistencies that exist among all latency benchmarks. Looking at Table 6.2 also aligns with this hypothesis, as we see more examples of relative trends.

6.2 MCDRAM/DRAM Bandwidth Results

Figures 5.1 through 5.9 show the results of the STREAM[12] Benchmark under the conditions mentioned in Section 4.7. Across all the benchmarks, we can see that the peak bandwidth measured using MCDRAM is in Figure 5.5, with a peak bandwidth of 400GB/s, similarly, the peak DRAM bandwidth measured is in Figure 5.4 (or alternatively at a scaled resolution, in Figure A.5), with a peak bandwidth of 70.5GB/s. Intel declared[17] that peak bandwidth for DRAM memory would be 90+ GB/s and peak bandwidth for MCDRAM memory would be 400+ GB/s under the STREAM's Triad 'operation'. Comparing to our peak results we see that we are able to achieve 400GB/s for MCDRAM, but only 70GB/s for DRAM under a Copy operation. There are two major observations here. Firstly; peak bandwidths were observed under the Copy operation, and Secondly; our peak bandwidths are slightly lower than those declared by Intel. Both of these observations are likely due to our test system being a 64 Core system, where there exists a Knights Landing model that contains 72 Cores. It is reasonable to believe that a 72 Core system would be able to obtain higher bandwidths, considering that more threads could be run simultaneously. That said, our measured bandwidths closely follow the expected trend; MCDRAM has `4x` the bandwidth of DRAM. Another trend in the results that is noteworthy, is we see the bandwidth increase up until 64 Threads, where we see the bandwidth either plateau or drop off. This makes sense, as our test system has 64 Cores, and there is an expected performance hit when multiple threads are running on a Core concurrently, sharing the Core's resources. We also see that in the case of Sub-NUMA Clustering, our peak measured bandwidths are roughly one-quarter of the peak measured bandwidth in `Quadrant` mode, which is the expected behaviour.

6.3 Summary

To summarise the results, it would be fair to say that the declared Bandwidths provided by Intel are accurate, and that bandwidths measured on our 64 Core test system would not achieve peak bandwidths relative to the 72 Core Knights Landing chips. When measuring latencies, it appears that the timing mechanism proposed by Intel[15] does provide a reliable timing mechanism for measuring code execution times, but perhaps loses accuracy when measuring load instructions. Had more time been available, further experimenting with the number of loads, or experimenting other memory access approaches could have been tried. For example, pointer chasing[3] is a common practice in benchmarking suites for generating random accesses to memory. When creating multi-threaded benchmarks for measuring coherence latencies, more work could have been done to ensure thread placement with respect to NUMA regions was resulting in the intended behaviour. This is especially important when considering data in the Shared state, which unfortunately provided no meaningful results following the benchmark structure outlined in Section 4.6 and Section 4.6.5.

[3]Pointer Chasing: Traversing a linked list, in order to achieve unpredictable memory accesses.

Chapter 7

Conclusion

7.1 Work Done

All code and algorithms presented within this report were my own work, with the exception of the STREAM benchmark, or other code snippets which were otherwise cited. The project work began with trying many different sample programs in order to confirm my understanding of computer architecture thus far. A large portion of the work that was done was into researching, trying, and evaluating possible options for measuring latency (as discussed in Section 4.1). After some base timing mechanism was established, I then worked on building algorithms to place data in specific areas of the memory hierarchy, and using the results and observations made, improvements were made to both the algorithms and the timing techniques. After obtaining measurements for all non-coherence related latencies, I primarily worked on researching how it is possible to measure the bandwidth of the DRAM/MCDRAM and reading Intel documentation on the subject. Many sources pointed to work conducted by John McCalpin, and I then worked on properly using the STREAM benchmark. The final chunk of work was looking into how to write multithreaded applications in the context of OpenMP but also simply c++ threading for the purpose of measuring Cache Coherence latencies. Due to the nature of the benchmark, it was important to have tightly synchronised threads, and so an OpenMP approach would be complicating the matter more than necessary. Using c++ threads, I worked on implementing an algorithm/program structure that would allow me to have multiple threads follow a single state machine. I then used work from my previous latency benchmarks to have threads/Cores perform actions on shared data such as reads/writes. Other work that was attempted but not fully explored includes techniques such as intentional cache thrashing to evict data from different caches, however, it did not provide immediate value to all the benchmarks written so it was not fully implemented. I also looked into using a pointer-chasing algorithm for generating random loads, but opted to try a different approach to measuring latencies relative to other benchmark programs.

7.2 Insight Gained

Throughout working on the project, I gained a lot of insight into how computer architecture is applied in a parallel fashion under the context of High Performance Computing. For example how HPC systems processor architecture is relevant to the data processed. The project also supplied a great opportunity to confirm my prior knowledge, for example how applications are executed from a high-level, down to a very low level. An example of this is that a lot of the work included inspecting assembly and understanding how compilers generate machine code, and how code is executed. I would also say that the project introduced me to some concepts of code optimisation, for example on the importance of writing cache-friendly code. Finally, the project also provided a great opportunity to solidify my understanding of concepts taught (specifically Cache Coherence) in the Parallel Architectures course taught by Vijay Nagarajan.

7.3 Project Outcome

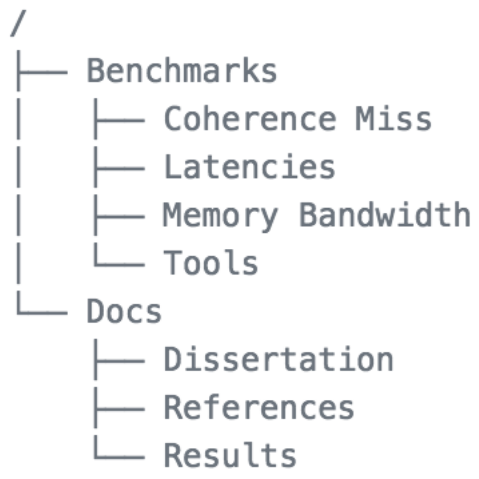
The outcome of the project presents a set of results for latencies and bandwidths of accessing data between different memory components of the Knights Landing processor. It is important to read the discussion of the results, as there is further room for improving the benchmark to obtain more accurate results. The project also delivers open-source benchmark programs that can be used for further research in the field. Had more time been available, further work into researching different ways of measuring latencies, and improvements on the current technique would be done. It would also be worth investigating cache thrashing, as an opportunity for placing data in specific areas of the Cache hierarchy.

Chapter 8

Open-Source Benchmark Suite

This Chapter provides no further discussion or evaluation of the work done in this project, but shows an outline of the benchmark produced. Specifically, this chapter will introduce how one could run the benchmarks for themselves. The same instructions, plus more background info, can be found at the GitHub repository root:
https://github.com/acwilson96/MicroBenchmarking-KNL

8.1 Repository Structure



```
/
├── Benchmarks
│   ├── Coherence Miss
│   ├── Latencies
│   ├── Memory Bandwidth
│   └── Tools
└── Docs
    ├── Dissertation
    ├── References
    └── Results
```

Figure 8.1: Repository Structure

8.2 Benchmarks

Contains source-code and associated files for building and executing the different microbenchmarks.

8.2.1 Coherence Miss

Contains source-code for the multi-threaded benchmark which evaluates latencies of accessing data across Cores.

Usage:

```
make
./run.sh State1Core ( M | E ) State2Core
./run.sh State1Core State2Core S State3Core
```

For example, if you wish to trial Core 5 accessing modified data in Core 8's L1 Cache, you would run `./run.sh 8 M 5`. Or if you wish for Core 1 to access Exclusive data in Core 0: `./run.sh 0 E 1`. Although the Shared benchmark is not complete, it can be run in a similar fashion, where if you want Core 6 to access data that exists in Core 9 and Core 15 you would run: `./run.sh 9 15 S 6`.

8.2.2 Latencies

Contains source-code for microbenchmark for evaluating L1 Cache, L2 Cache, and DRAM/MCDRAM access Latencies.

Usage:

```
make
./run.sh numa_region
```

Where numa_region is the numa-region you wish to run your benchmarks on. In Quadrant/All-to-All and Flat Mode, 0 is DRAM, and 1 MCRAM. It is recommended to perform latency measurements in Quadrant/All-to-All, as the benchmark will interpret the ”0” or ”1” passed as an argument, to alter the output results.

8.2.3 Memory Bandwidth

Contains source-code for STREAM Benchmark purposed for Knights Landing. STREAM Courtesy of John McCalpin @ https://www.cs.virginia.edu/stream/

Usage:

```
make
./run.sh numa_region
```

Where numa_region is the numa-region you wish to run your benchmarks on. In Quadrant/All-to-All and Flat Mode, 0 is DRAM, and 1 MCRAM.

8.2.4 Tools

Contains small utility application for converting Cycles to NanoSeconds

8.3 Docs

Contains source-control files for the dissertation and results associated with the dissertation.

Bibliography

[1] numactl man page. https://linux.die.net/man/8/numactl.

[2] numactl repository. https://github.com/numactl/numactl.

[3] Various Authors. BenchIT Microbenchmark Suite. https://tu-dresden.de/zih/forschung/projekte/benchit/?set_language=en. Last Accessed: 2018-03-01.

[4] StackOverflow User: Ankur Chauhan. Corepin() source. https://stackoverflow.com/a/43778921, 2017. Last Accessed: 2018-03-01.

[5] Wolfgang E. Nagel Daniel Hackenberg, Daniel Molka. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. https://tu-dresden.de/zih/forschung/ressourcen/dateien/abgeschlossene-projekte/benchit/2009_MICRO_authors_version.pdf?lang=en.

[6] Robert Schone Wolfgang E. Nagel Daniel Molka, Daniel Hackenberg. Cache coherence protocol and memory performance of the intel haswell-ep architecture. https://tu-dresden.de/zih/forschung/ressourcen/dateien/abgeschlossene-projekte/benchit/2015_ICPP_authors_version.pdf?lang=en, 2015.

[7] Agner Fog. Instruction Tables, Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs. http://www.agner.org/optimize/instruction_tables.pdf, 1997-2017.

[8] Eric G. What public disclosures has intel made about knights landing? https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing, 2016.

[9] StackExchange User: Gilles. /proc/cpuinfo Flags. https://unix.stackexchange.com/a/43540. Last Accessed: 2018-03-01.

[10] jain.pk. Using inline assembly in c/c++. https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C. Last Accessed: 2018-03-01.

[11] Ph.D. John D. McCalpin. Stream faq. https://www.cs.virginia.edu/stream/ref.html#what.

[12] Ph.D. John D. McCalpin. Stream memory bandwidth in hpc. https://www.cs.virginia.edu/stream/.

[13] John McCalpin. Memory latency on the intel xeon phi x200 “knights landing” processor. https://sites.utexas.edu/jdm4372/2016/12/06/memory-latency-on-the-intel-xeon-phi-x200-knights-landing-processor/, 2016.

[14] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[15] Gabriele Paoloni. How to benchmark code execution times on intel® ia-32 and ia-64 instruction set architectures. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf, 2010. Last Accessed: 2018-03-01.

[16] Florian R. Measuring performance in hpc. https://software.intel.com/en-us/articles/measuring-performance-in-hpc. Last Accessed: 2018-03-01.

[17] Avinash Sodani. Knights landing (knl): 2nd generation intel® xeon phi™ processor. Mirror: https://github.com/acwilson96/MicroBenchmarking-KNL/blob/master/Docs/References/HC27.25.710-Knights-Landing-Sodani-Intel.pdf, 2016.

[18] http://www.felixcloutier.com. CPUID Instruction Description. http://www.felixcloutier.com/x86/CPUID.html. Last Accessed: 2018-03-01.

[19] http://www.felixcloutier.com. MFENCE Instruction Description. http://www.felixcloutier.com/x86/MFENCE.html. Last Accessed: 2018-03-16.

[20] http://www.felixcloutier.com. RDTSCP Instruction Description. http://www.felixcloutier.com/x86/RDTSCP.html. Last Accessed: 2018-03-16.

Appendix A

Appendices

A.1 Timestamp Counter Support

On Linux, you can find out if your CPU supports invariant-tsc by running the following command on the command line:

```
cat /proc/cpuinfo
```

This will list information about the Processors that the OS is aware of, and for each visible Processor, will include a list of ”flags” that the Processor supports.

The flags[9] of interest for Intel systems under this project are:

- **`tsc`**
  Core(s) Supports a Time-Stamp-Counter mechanism that can be read using the `RDTSC` Assembly Instruction.
- **`constant_tsc`**
  The TSC ticks at a constant rate.

A.2 Results

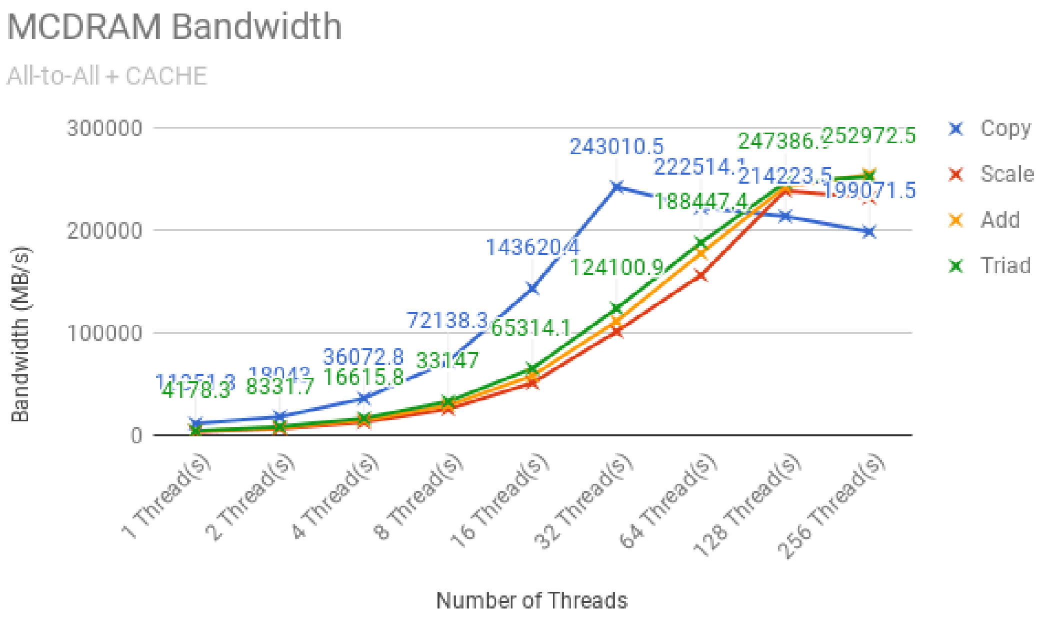


Figure A.1: MCDRAM - Cache Mode - Scaled

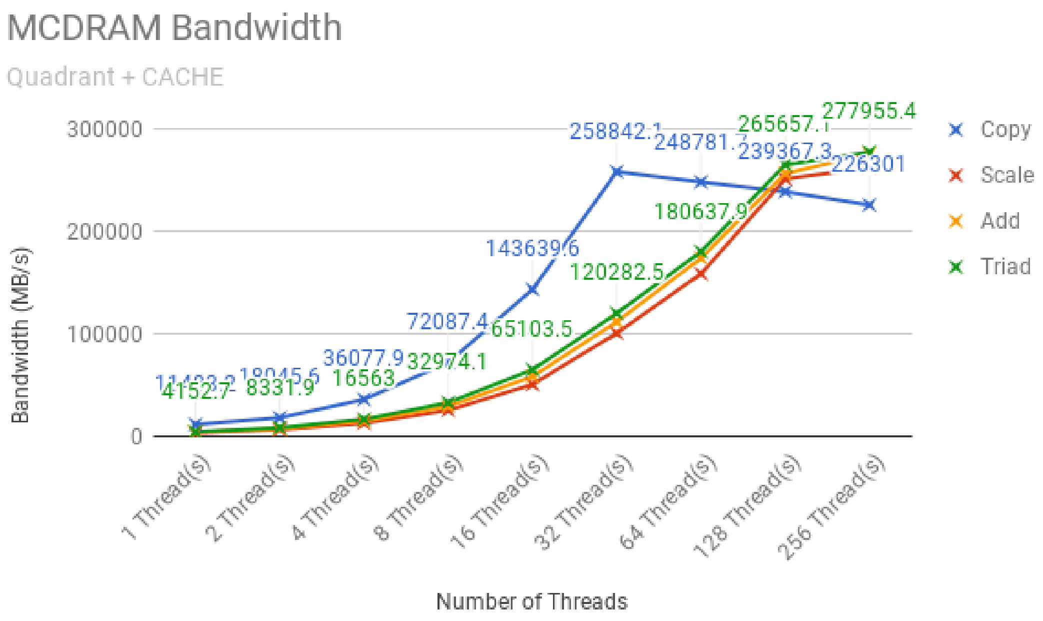


Figure A.2: MCDRAM - Cache Mode - Scaled

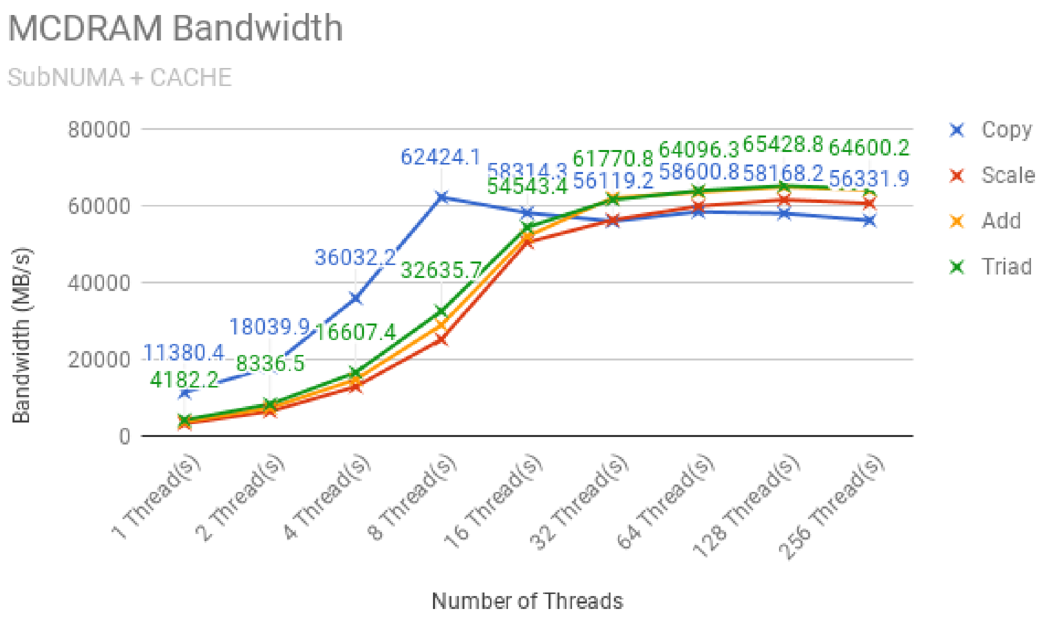


Figure A.3: MCDRAM - Cache Mode - Scaled

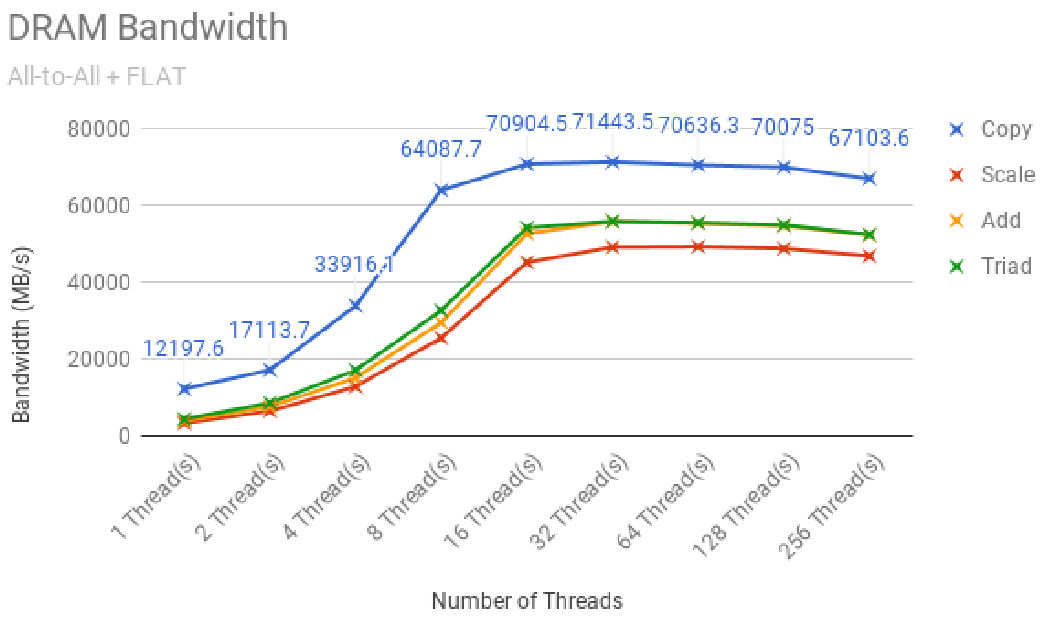


Figure A.4: DRAM - Cache Mode - Scaled

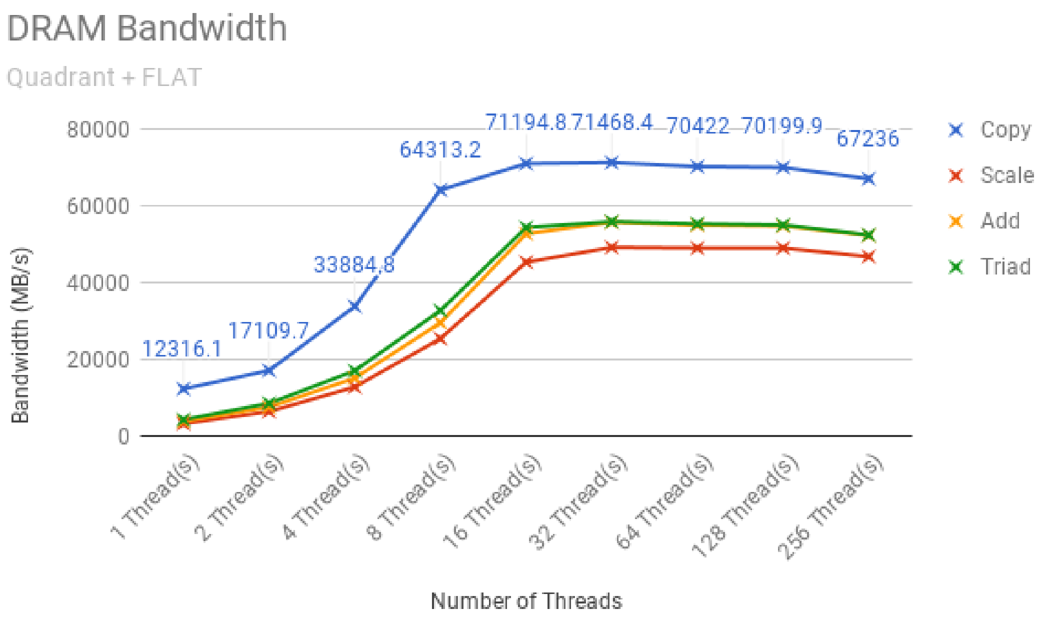


Figure A.5: DRAM - Cache Mode - Scaled

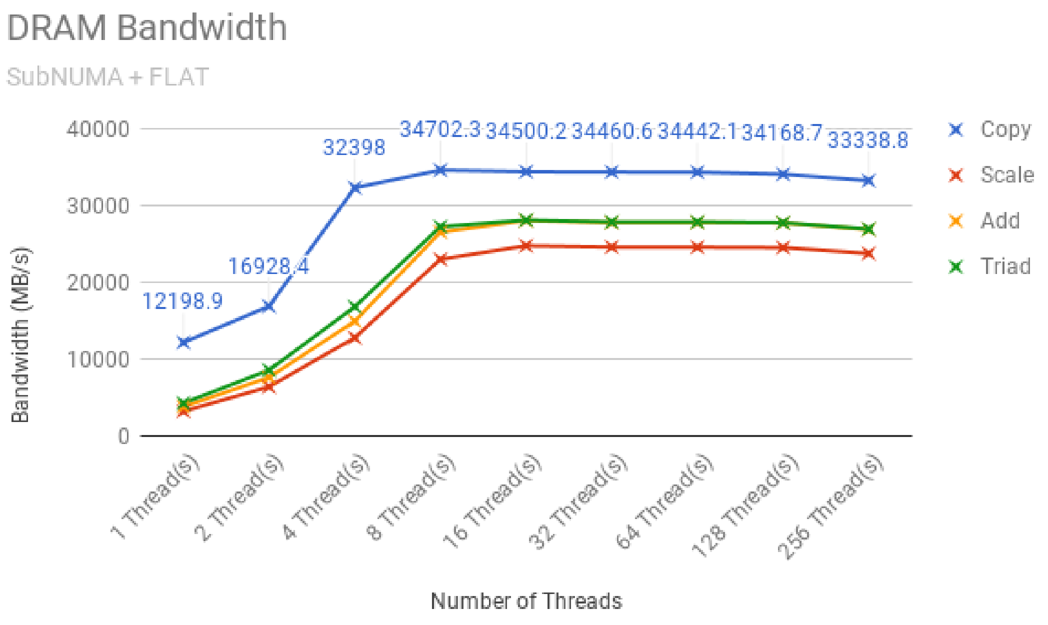


Figure A.6: DRAM - Cache Mode - Scaled