Slice-Aware Memory Allocation

A programmatic approach to reduce LLC access latency

Henrik Bjørseth, Nathanael Eneroth,
Jimmy Fjällid, Ali Shokrollahi, Farhad Zareafifi

IK2200 Project Report

Communication Systems
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden

6 January 2019

Examiner: Professor Dejan Kostic
Supervisor: Alireza Farshin

Abstract

Technology is continuously improving and becoming more efficient. Network speeds are increasing and memory sizes are getting larger. As a result, more computational power is required from the Central Processing Unit (CPU). The increase in network capacity is encouraging the development of more efficient software applications to fully utilize on fast networks. However, developing efficient software requires a deep understanding of hardware, and more specifically, a deep understanding of memory management. Thus, knowledge of the memory hierarchy enables further optimization of applications.

The CPU stores recently accessed information in caches located close to the CPU cores. This project shows that access time to different parts of the Last Level Cache (LLC) varies depending on which core an application is using. For this reason, it proves to be beneficial to restrict memory usage for each core, and to force each core to only utilize a part of LLC memory with minimal latency.

This paper presents a library that measures memory access time to different cache levels and to main memory. The library can also restrict memory allocation to specific parts of LLC. More specifically, the library can benchmark normal memory allocation against an allocation scheme that only uses LLC parts with minimal latency (slice-aware memory allocation). The results show that performance can be improved with slice-aware memory allocation when an application's working set can fit in LLC.

Keywords: Linux, CPU, cache, LLC, Intel, Haswell, memory latency.

Sammanfattning

Tekniken utvecklas kontinuerligt och blir mer effektiv. Nätverkshastigheterna ökar och minnesstorleken växer. Denna utveckling leder till att bättre prestanda krävs från en processor (CPU). Höga nätverkshastigheter uppmuntrar till utveckling av snabbare mjukvara som kan nyttja höghastighetsnätverk, men för att utveckla effektiv mjukvara krävs en djup förståelse i minneshantering. Av denna anledning utgör effektiv minneshantering en nyckelaspekt i utvecklingen av bättre mjukvara.

En processor lagrar nyligen använt minne i cachelager placerade nära en processors olika kärnor. Det här projektet visar att åtkomsttiden till det sista cachelagret (LLC) varierar beroende på vilken kärna en applikation utnyttjar. Detta medför att det är förmånligt att nyttja LLC-minne med minimal latens till respektive kärna.

Det här projektet presenterar ett bibliotek som kan mäta latenstiden till olika cachelager samt till huvudminne. Biblioteket kan också begränsa minnesallokering till specifika delar av LLC, och mäta skillnaden mellan normal minnesallokering mot allokeringsmetoden som endast nyttjar LLC minne med minimal latens. Resultatet visar att prestandan förbättras avsevärt med den senare allokeringsmetoden när en applikation primärt nyttjar LLC minne.

Nyckelord: Linux, CPU, cache, LLC, Intel, Haswell, minneslatens.

Acknowledgements

We would like to express our gratitude to our supervisor Alireza Farshin for his help and guidance throughout this project.

Stockholm, January 6, 2019

The MEM group

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms and Abbreviations

CPU	Central Processing Unit
DRAM	Dynamic Random-Access Memory
HDD	Hard Disk Drive
L1	Level 1
L2	Level 2
L3	Level 3
LLC	Last Level Cache
LRU	Least Recently Used
MMU	Memory Management Unit
MOPS	Mega Operations Per Second
OPS	Operations Per Second
OS	Operating System
PMON	Performance Monitoring
SKU	Stock Keeping Unit
SSD	Solid-State Drive
TLB	Translation Lookaside Buffer

Chapter 1

Introduction

There has been unprecedented growth in network speed and network capacity. However, the link speed is limited by inefficient utilization of the Central Processing Unit (CPU) and the gap between link speeds and CPU efficiency keeps increasing. CPUs have more computational power due to more cores and higher frequencies. The last decade has not seen any significant increase in CPU frequency which limits single core performance. To circumvent this, focus has been directed towards adding more CPU cores to increase performance [2]. Moreover, memory has also become larger and faster. These hardware advancements open up for high-speed applications which are able to deliver more predictable response times. However, the performance of these applications depends not only on good programming practice but also on efficient memory management. Without considering memory access times, applications are unable to deliver their best possible performance. Thus, it is important to understand how an application's performance is related to memory management of specific hardware. For this reason, the underlying hardware needs to be considered to further improve performance and system latency.

1.1 Overview

Despite the fact that technological advancements improve memory latency, these advancements do not keep up with CPU performance. The CPUs are faster and are improving at a quicker rate compared to memory [3]. However, memory can be created to be fast enough to keep up with the CPU, but it is prohibitively expensive to use this fast memory in large enough quantities to hold the working set in memory. A compromise is to use a smaller number of very fast memory caches located very close to the CPU cores. In this way, multiple levels of caches

are added between the CPU and main memory; hence the performance gap can be decreased. These caches are of increasing size but with slower speed as they approach the main memory. In order to reduce the number of times the CPU must access main memory, recently used data is stored in a hierarchy of cache levels [3].

Caches can be categorized into three different levels: Level 1 (L1), Level 2 (L2), and Level 3 (L3), also known as Last Level Cache (LLC). Figure 1.1 illustrates how different cache levels are connected to various CPU cores in Intel's Haswell architecture. L1 and L2 caches are small, very fast, and private to each core. However, LLC cache is shared among all cores and it is typically larger than L1 and L2 cache [1]. In contrast to L1 and L2, LLC is divided into slices. As illustrated in Figure 1.1, the CPU cores are connected to LLC slices via a ring bus; hence, a core has different access latency to each LLC slice. Thus, each core has one LLC slice with minimal access latency.

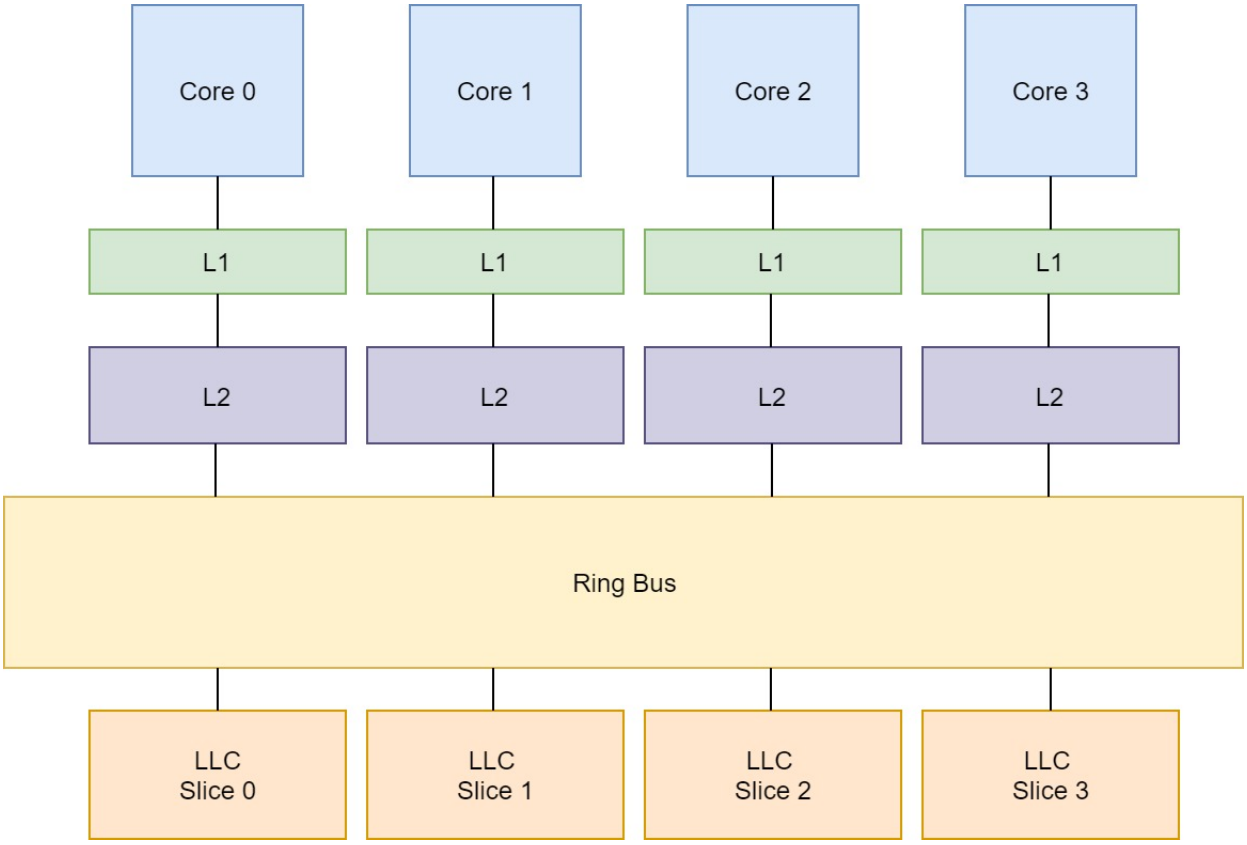

Figure 1.1: Cache architecture of a Quad-Core Intel Haswell CPU, adapted from [1].

1.2 Problem

Many applications suffer from inefficient memory management since CPU cores do not exclusively use LLC memory with minimal latency. Instead, each core randomly selects a LLC memory address to use; hence, an application utilizes memory with both high and low access latency. Thus, high speed applications that primarily utilize LLC memory are unable to deliver their best possible performance. Currently, there exists no available tool or library, known to the authors, that can measure access time to individual LLC slices.

Hypothesis: Minor improvements in memory management can greatly enhance the performance of high-speed applications.

1.3 Purpose

The purpose of this project is to develop a C++ library that can measure access latency to different levels of the memory hierarchy, i.e., L1, L2, LLC, and Dynamic Random-Access Memory (DRAM). Moreover, this library should provide the means to compare LLC slice-aware memory allocation vs random (standard) memory allocation. In this way, this library will illustrate how slice-aware memory allocation performs next to random memory allocation. In this project, slice-aware memory allocation is defined as exclusively using memory from a single LLC slice with minimal latency.

1.4 Goals

The goal of this project is to improve application performance by:

1. Studying the memory organization of an Intel Haswell CPU in a Linux-based operating system.

2. Developing an architecture-aware library for measuring access times to different cache levels and DRAM.

3. Developing a new memory management scheme to exploit slice-aware memory allocation.

4. Comparing slice-aware memory allocation with random memory allocation.

1.5 Deliverables

- The git repository [4] containing the developed library.
- Project plan
- Midterm presentation of the progress so far.
- The final report
- Oral presentation

1.6 Research Methodology

This project employs a quantitative approach [5] since the main purpose of this project is to prove, through experiments, that slice-aware allocation can decrease memory access times for delay sensitive applications.

1.7 Delimitations

This project only focus on memory organization in the Linux operating system using an Intel CPU. Thus, any other CPU architectures (e.g., AMD or ARM) and other operating systems (such as Windows or Mac OS) are not included in this project. The library is architecture-aware, but it is currently limited to work solely on Intel's Haswell architecture. This project only considers L1, L2, LLC, and DRAM for measuring memory access times. Moreover, all the experiments for slice-aware memory allocation are performed on an isolated CPU socket.

1.8 Report Structure

The rest of this report is organized as follows: Chapter 2 contains background information required to understand this project. The Methodology chapter, Chapter 3, presents the research methodology of this project. Chapter 4 describes the implementation of the measurements performed in this project. Chapter 5 presents the results and analysis. Chapter 6 concludes this project with discussion and future work.

Chapter 2

Background

This chapter provides background information to understand general memory management present in Intel's CPU architecture.

2.1 Virtual and Physical Memory

In computing, virtual memory gives applications the illusion that they have access to a continuous large block of memory. However, in reality, physical memory is fragmented, and because of this, virtual memory solves three potential problems [6]:

1. More memory is needed than what is physically available.
2. Gaps in the address space created by terminated applications.
3. Processes overwriting memory addresses of other processes.

Virtual memory management makes it possible for an application to utilize memory that exceeds the physical limitation of a system, i.e., virtual memory enables mapping of memory resources to a physical hard drive. When multiple processes share the same physical memory, gaps of unused memory will appear if some of them are terminated. If each process operates with the impression of exclusive memory access, on the other hand, this may conflict with other processes utilizing the very same physical memory. One process may write into the same memory location as another process, thereby overwriting information used by other applications. For this reason, virtual memory is used to ease memory management when multiple applications are running concurrently. A process request blocks of virtual memory that is mapped to blocks of physical memory. The mappings between virtual and physical memory blocks are stored in a page table [7].

2.2 Translation of Virtual to Physical Memory

The process of virtual to physical address translation is done by the Memory Management Unit (MMU). MMU is usually a part of the CPU and handles all operations related to memory management [8]. Therefore, its main responsibility is to translate each virtual address to a corresponding physical address. Most MMUs use a page table for this address translation. When a process requests memory, it is assigned a virtual page which contains a set of contiguous memory addresses. Virtual pages are mapped to physical page frames, i.e., sets of contiguous physical addresses. These mappings are stored in a page table located in memory. Figure 2.1 illustrates the conversion of a virtual address to a physical address. The MMU uses a virtual page number as an index to the page table, and the page frame number is concatenated with a page offset to create the physical address. The complete page table reside in main memory, but to reduce access times for memory translation, a cache memory is used as buffer. This cache memory is known as a Translation Lookaside Buffer (TLB) and is part of the MMU [9]. TLB stores recent memory translations, hence it is not necessary to access main memory if a memory translation reside in TLB.

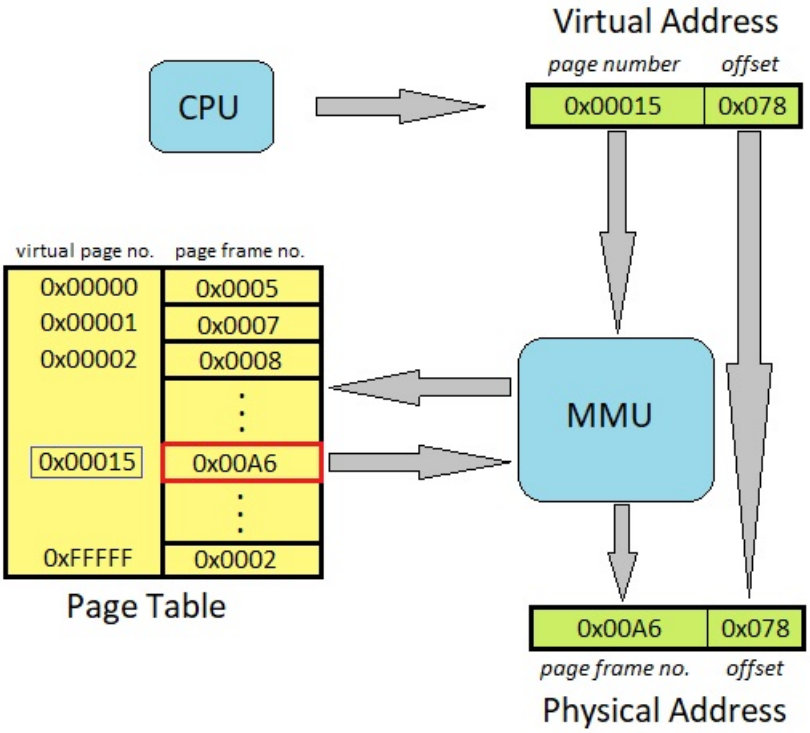

Figure 2.1: Virtual to physical address translation.

2.3 Cache Fundamentals

Modern Intel CPUs store recently used data in different cache levels to reduce access latency and to increase application performance. L1 cache is the smallest and fastest cache located closest to the CPU. It is typically not larger than 64 KiB in size, and is split in two equally sized parts for data and CPU instructions, respectively. The size and access time of each cache level varies between CPU architectures, but recent Intel CPUs have a L1 cache access latency of 4 CPU cycles [10]. The next cache level is L2 cache. L2 cache is commonly between 256 KiB and 1 MB in size for pre-Skylake and post-Skylake chips, respectively, and has an access time of 7 CPU cycles [11]. The first two cache levels are private to each core, i.e., each core has its own dedicated L1 and L2 cache. The third and last cache level is LLC. LLC is slower than both L1 and L2 cache, but faster than main memory. LLC size varies from 3 MB to over 20 MB, hence LLC has different access latency, typically between 26 and 31 cycles [12]. Unlike L1 and L2 cache, LLC is shared among all cores of the CPU and is divided into multiple slices. The number of LLC slices correspond to the number of cores in the CPU. The slices themselves are interconnected to the CPU cores, e.g., via a ring bus in Haswell.

As illustrated in Figure 2.2, each cache level is divided in sets and lines (ways), each line is 64 bytes. The number of sets depends on cache size, and the number of lines in a set depends on associativity.

$$\text{CacheSize} = \text{NumOfSets} \times \text{Associativity} \times \text{LineSize}$$

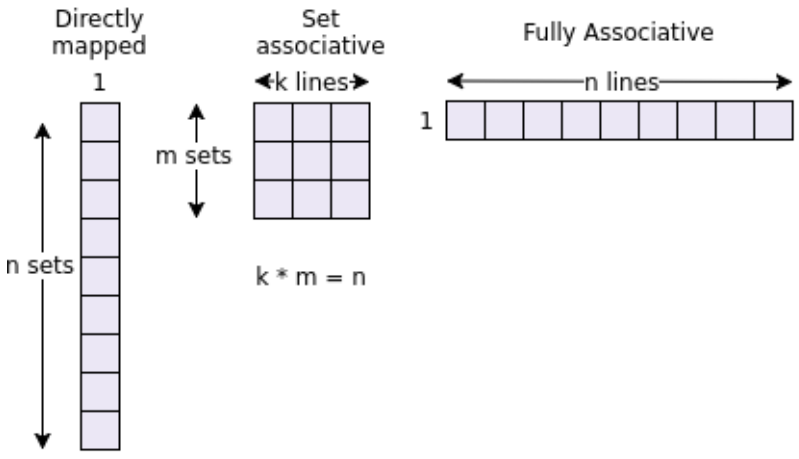

Figure 2.2: Cache associativity.

Associativity is described as n-way associativity, where n signifies the number of lines in a set. This means that data written to a physical memory address can be stored in any of the n cache lines. There are three types of associativity:

- Directly mapped (1-way)
- Set associative
- Fully associative

In directly mapped cache, there is one cache line per set and each physical address only maps to one cache line. Thus, directly mapped cache is represented as an $n \times 1$ matrix where n is the number of sets.

In set associative cache (Figure 2.3), there can be m sets where each set contains k lines. Every line in a set maps to the same physical address*. For example, if k=8, the cache is 8-way associative, i.e., there are 8 cache lines per set, and each cache line is typically 64 bytes in size. Thus, a 32 KiB L1 data cache, which is 8-way associative, has $8 \times 64 = 512$ bytes in each set. The number of sets available in cache memory is calculated by dividing the cache size with the set size. For this reason, a 32 KiB cache will contain $\frac{32768}{512} = 64$ sets.

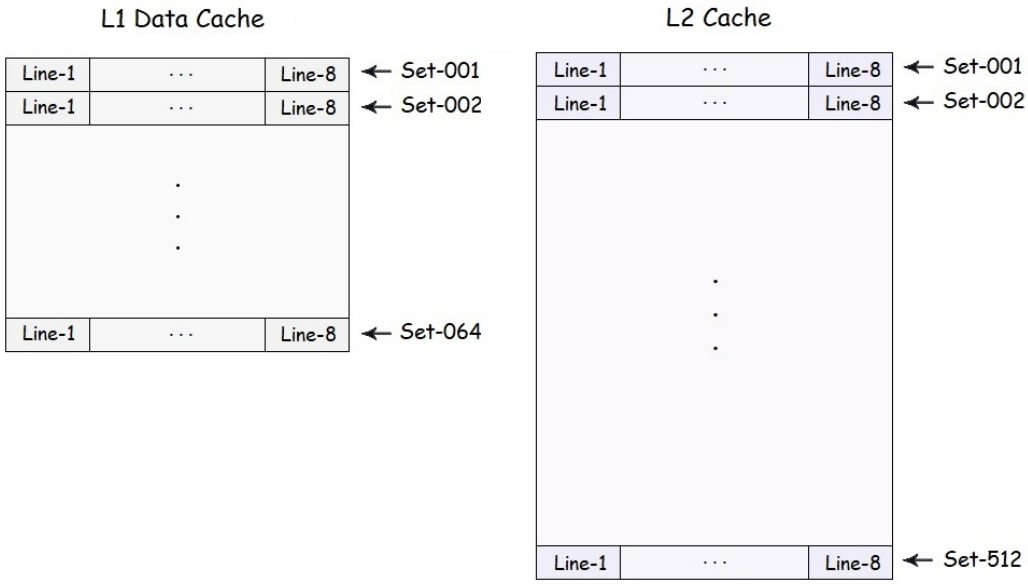

Figure 2.3: 8-way associative L1 and L2 cache structure.

* I.e., the content of a memory address could be cached in any of the lines in a set.

In fully associative cache, there is only one set but n lines. Hence, each physical address maps to the same set. Figure 2.3 illustrates L1 and L2 cache that are 8-way associative.

2.3.1 Comparison of Cache Associativity

Directly mapped cache is more power efficient [13, 14] than set associative and fully associative cache, and has a simple replacement policy. Unfortunately each physical address is mapped to only one cache line and hence the probability of a cache miss* is high. Fully associative cache, on the other hand, maps each physical address to one set. This is very expensive since each cache line must be examined for every read operation to locate the line an address points to. However, fully associative cache better utilizes the entire cache since any cache line can be used. The compromise is to use set associative cache, i.e., a trade-off between efficiency and cost (execution time and power consumption)

The cache replacement policy dictates which cache line that should be evicted in order to make room for new blocks of data. A common practice is to use an approximation of the Least Recently Used (LRU) policy [15]. Thus, the cache line that was least recently used is evicted since it has the lowest probability of being used again [15].

2.4 LLC Slice Mapping

LLC is divided in different slices which are shared among the CPU cores. The number of slices are equal to the number of cores. Similar to the other cache levels, every slice is divided into different sets.

There are different schemes to map physical addresses to LLC slices. The most basic scheme is *direct addressing* where each physical address is directly mapped to a specific LLC memory. Therefore, this approach is very simple, but unfortunately easy to compromise in terms of security. For this reason, critical information such as cryptographic keys could get compromised [16] in an attack which targets a specific set in LLC. One well-known attacks is *prime+probe* attack [17]. In this attack, the attacker fills the whole cache with data and waits for cache sets to be evicted. Information about evicted cache sets enables the attacker to retrieve the victim's private information. In order to prevent this type of attack, the slice each physical address maps to must be hidden.

* When the processor finds that the memory location is in the cache it is known as a cache hit. If the memory location is not located in the cache it is known as a cache miss.

Complex addressing is a scheme that hides the mapping between physical addresses and LLC slices. This makes it harder for an attacker to determine the slice to which a physical address is mapped. In a *complex addressing* scheme, each physical address has three different parts: tag, set, and offset. Offset includes the lowest bits of a physical address and is used to identify individual bytes of a cache line, while set bits determine the cache set of an address. However, set bits are also used along with tag bits to identify a LLC slice.

2.4.1 Hash Function

A hash function is used to determine the mapping between a slice and a physical address [1]. This hash function applies to Haswell Intel CPUs and is used in experiments throughout this project. In complex addressing, the hash function takes set and tag bits as input and returns the slice number of a physical address. Figure 2.4 illustrates complex addressing, and how the hash function determines the slice number.

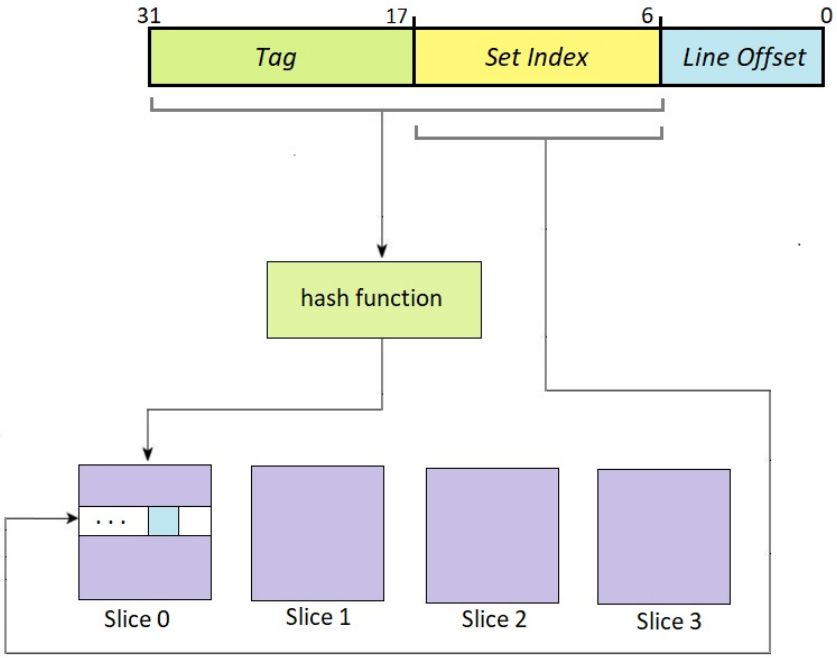

Figure 2.4: Hash mapping to LLC slices, adapted from [1].

2.4.2 Uncore Performance Counters

Intel CPUs have a set of special purpose registers external to the CPU called uncore performance counters. These can be used to measure global events in the CPU such as cache misses and branch mispredictions - they can also be used for benchmarking. The counter registers are paired with control registers to select which type of event that should be counted. These registers are grouped in Performance Monitoring (PMON) units, more specifically, there is one PMON called a C-BOX. There is one C-BOX for every LLC slice, and the counters in the C-BOX can be configured to count all LLC access events [18]. For this reason, C-BOX counters can be used to determine which LLC slice a physical address belongs to by polling an address multiple times. The C-BOX counter with highest value identifies the LLC slice a physical address is mapped to, i.e., only the C-BOX counter in the correct LLC slice increments when a memory address is accessed. Thus, polling that address a high number of times will clearly distinguish which LLC slice the address maps to.

2.5 User Space and Kernel Space

Memory can be divided into two different regions in the Linux Operating System (OS), kernel space and user space. Kernel space memory is, as the name implies, allocated for the kernel and can only be accessed from user space through the use of system calls [18]. In this way, the kernel will isolate a part of system memory which user space applications can not access.

User space memory is used by all other processes (a part from the kernel). The kernel is assigned to monitor user space memory and to make sure that processes in user space do not interfere with each other [18]. In terms of this project, memory measurements will be performed in user space. User space measurements will always be exposed to noise since the kernel is responsible for memory management.

2.6 CPU Socket Isolation

In order to give a process as much execution time as possible, a CPU core can be isolated from the OS*. In this way, the OS will not assign any user-space threads or unbound kernel threads to the isolated CPU core [19, 20]. More importantly, core isolation also prevents a core from being interrupted, and hence it is possible to avoid interference by other processes. As illustrated in Figure 1.1, LLC is shared

* Provided there are more than one CPU core.

among all CPU cores. Isolating a CPU core will isolate the L1 and L2 cache, but the LLC can still be accessed by the other cores. To perform undisturbed LLC measurements, all cores must be isolated in a socket. Thus, a system that supports multiple CPU sockets is required.

2.7 Related Work

Yuval Yarom et al [10] developed a technique for reverse-engineering the Intel hash function. They applied this technique for a 6-core Intel processor and used it for cache-based side channel attacks. They also used it to double the number of colors used for page-coloring techniques.

Olivier Heen et al designed a method for reverse engineering Intel's LLC complex addressing [1]. In the paper, the authors determine which LLC slice an address maps to by using performance counters.

There are other memory benchmark tools that can measure access latency to CPU caches and DRAM. One such tool is LMbench [21] which is a versatile performance monitoring tool written by Larry McVoy and Carl Staelin. While LMbench can measure cache access latency, it can not distinguish between different LLC slices which is necessary to prove this project's hypothesis.

Chapter 3

Methodology

This chapter provides an overview of the research method used in this project. Section 3.1 describes the research process, and Section 3.2 explains the data collection technique used throughout the project. In Section 3.3, the experimental design of the project is presented, while Section 3.4 explains validity and reliability of the collected data.

3.1 Research Process

The various steps carried out in this research project are listed in order in Figure 3.1. This project was initiated by conducting background research on CPU cache hierarchy, virtual to physical memory address translation, and allocation of memory pages. After the necessary background knowledge was acquired, a library was developed to perform some basic measurements. These basic measurements include measuring access time for L1, L2, and DRAM. The measurements were performed by reading some data into (specific parts of) memory, and then reading data from a specific memory layer while measuring the number of cycles required to perform the read operation. The next step was to isolate one of the two CPU sockets. This was done in order to reserve the entire LLC of the isolated CPU socket. After isolating a CPU socket, the next step was to extend the library to measure access times of different LLC slices. All of the LLC measurements were performed on the isolated CPU socket. In the final step, the library was extended to include benchmarking of slice-aware vs random memory allocation.

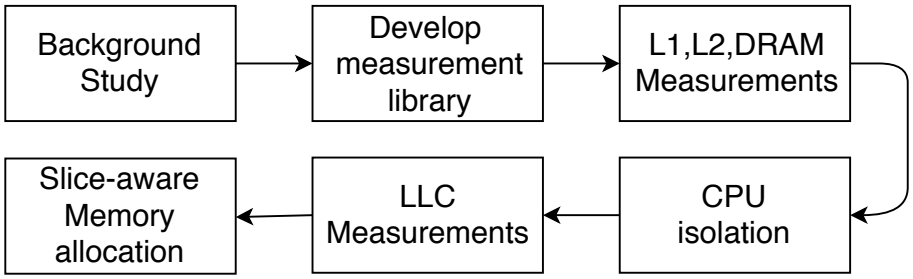

Figure 3.1: Research process steps

3.2 Data Collection

The data collected in this project was the access times for different cache levels and DRAM. The technique used for these measurements was to fill a specific part of memory with data, and then read from this memory layer while measuring the number of cycles required to perform the read operation.

3.3 Experimental Design

This section explains the environment experiments in this project were carried out in. This includes software and the specific hardware used throughout the measurements.

3.3.1 Testbed

The programming languages used throughout this project are C and C++. Python is used for automation. A GitHub repository was created for code version control and code reviews.

3.3.2 Hardware Platform

This project utilize one x86_64 physical machine with two different CPU sockets and sixteen CPU cores - eight cores per CPU package was used. The CPU model was Intel Haswell Xeon E5-2667 v3 3.20GHz. Thus, the size of L1 and L2 cache was 64 KiB and 256 KiB respectively, and the size of LLC was 20480 KiB.

3.3.3 Software Platform

The operating system used throughout the experiments in this project was Ubuntu 16.04.5 LTS (Xenial Xerus). Matlab and gnuplot was both used to generate data

plots. The library developed in this project was used to perform measurements (to collect data) of different cache levels.

3.4 Reliability and Validity

This project required to validate measurements before moving forward to the next step. For this reason, collected data had to be reliable and valid. The following subsection states how both reliability and validity were assured.

3.4.1 Reliability

Every measurement was repeated 10 000 times to minimize the impact of occasional noise. This number was experimentally determined to produce confident results while the measurements completed in a reasonable time. A machine with dual CPU sockets and two CPU packages was used to isolate one CPU package from the OS. The measurements were performed on the isolated CPU to prevent other processes from interfering with measurements, and to reserve the entire LLC.

3.4.2 Validity

Intel published a datasheet containing known results [12]. The validity of data collected in this project were verified by comparing measurements to this datasheet.

Chapter 4

Implementation and Measurements

This chapter explains the approach to measure access latency to different cache levels, and the measurement tool developed to perform these measurements. Section 4.1 explains measurements of L1, L2, and DRAM. Section 4.2 describes virtual to physical address translation. Section 4.3 describes LLC measurements. Section 4.4 defines Multithreaded Slice-aware Allocation. Finally, Section 4.5 briefly explains the developed measurement library.

4.1 Measurements of L1, L2, and DRAM.

The approach to measure cache latency is similar to all cache layers. It is necessary to read more data into memory than what would fit in the cache level below, and then read some of it again and measure how many clock cycles were required for each read operation. The measurements were put inside a loop made to perform the same measurement 10 000 times*.

The means to measure CPU cycles required for an operation to execute is presented in [22]. This approach relies on set of serialized assembly instructions that measures the cycle count before an operation executes, and after an operation has finished executing. For simplicity, these assembly instructions are omitted from the pseudo code examples throughout this chapter, but interested readers should consult [22] for further information.

* Occasional noise has minimal impact when the data set is large. Thus, it is necessary to perform several measurements to attain high confidence in sample data.

4.1.1 CPU Isolation

The experiments in this project were performed on an isolated CPU socket in order to avoid user space processes from impacting the measurements. When a CPU core is isolated, the kernel will no longer assign any processes to execute on that CPU core. However, a process can be manually or programmatically pinned to an isolated CPU core, such as the measurements performed in this project.

It is necessary to modify a boot parameter in the bootloader GRUB to isolate a CPU socket from the physical machine running Ubuntu 16.04. The parameter *isolcpus* specifies the CPU cores that should be isolated during boot-up. In order to identify the CPU cores that belongs to a physical CPU socket, the *lscpu* command can be used. Listing 1 shows the output of lscpu on a system operating with two Xeon E5-2667 v3 packages with 8 CPU cores.

```
NUMA node0 CPU(s):     0-7
NUMA node1 CPU(s):     8-15
```

Listing 1: CPU Core assignment per socket.

Listing 1 makes it clear that core 0-7 were mapped to node 0. Thus, core 0-7 were isolated using the GRUB parameter isoplus shown in Listing 2.

```
GRUB_CMDLINE_LINUX="isolcpus=0,1,2,3,4,5,6,7\
```

Listing 2: Option in Grub to isolate CPU cores at boot.

The measurement program is then manually assigned to run on one of the isolated CPU cores. This is done by setting the CPU affinity with the program *taskset*, or by programmatically assigning a CPU core an application should execute on.

4.1.2 L1 Cache Measurements

L1 cache measurements are the simplest measurements to perform. Assuming a LRU cache eviction policy, reading the last value written to L1 cache should guarantee a cache hit. For this reason, it is enough to fill the cache with data and then measure the number of cycles required to read the last value. Listing 3 shows pseudo code of L1 cache measurements.

```
void measure_l1()
{
   char* addr = allocate(L1_size);

   for(unsigned int i=0; i < L1_size; i++)
   {
      addr[i] = 1;
   }

   for(int k=0; k<N_MEASUREMENTS; k++)
   {
      start_time = cycles::now();
      char some_data = addr[0];
      stop_time = cycles::now();

      save_to_file(stop_time-start_time);
   }
}
```

Listing 3: Pseudo code of L1 measurements.

4.1.3 L2 Cache Measurements

To measure L2 cache latency, precautions must be taken to make sure that data to read is only present in L2 cache but not in L1 cache. Three different approaches to measure L2 cache were evaluated in this project:

- Allocate the entire L2 cache
- Access L2 cache using index bits
- Allocate twice the L1 cache size

The first approach fills the entire L2 cache with data and then reads some data that presumably is present in L2 cache but not in L1 cache. Since L1 cache contains the most recently accessed memory addresses (and since L1 cache can only store 32 KiB of data), reading from a memory address that is not among the last 32 KiB of accessed data should guarantee a L2 cache hit (and a L1 cache miss). To select an address that is not among the last 32 KiB of accessed data, a pointer with a negative offset of 32 KiB to the last address accessed is used. This pointer would thus point to a memory location that is just before the last accessed 32 KiB; hence this memory address is not present L1 cache but presumably present in L2 cache.

However, this approach often produced incorrect results, i.e., more CPU cycles than what is theoretically justified was required to access L2 memory. Even though some attempts would yield the correct results, most of the measurements were too high. This suggests that the address that was assumed to be in L2 cache was sometimes only present in the LLC.

The second approach exploits the knowledge that there are index bits in a physical memory address that decides which cache set an address can be mapped to. As explained in Section 2.3, the L1 cache is 8-way associative. Assuming a 64 byte cache line and a 32 KiB cache size, the number of sets in the L1 cache is equal to:

$$\frac{32\text{KiB}}{8 \times 64} = 64 \text{ sets}$$

Since L2 cache is eight times larger than L1 cache but with similar associativity, L2 cache contains eight times more sets than L1 cache. Moreover, multiple sets in L2 maps to the same set in L1. For example, set 0 and set 64 in L2 cache both map to set 0 in L1 cache. Thus, accessing 8 addresses in set 0 will cache them in L1. However, when 8 addresses of set 64 is accessed, they will be evicted from L1 since set 0 and set 64 addresses can not occupy the same set at the same time (in L1 cache). Thus, accessing an address from set 0 again should for this reason guarantee a miss in L1 cache but a hit in L2 cache. Using this knowledge, it is enough to only access these two sets for L2 measurements. However, this approach would yield results that were too low, i.e., the number of cycles reported was lower than expected. It is likely that this behaviour occurs due to hardware prefetching, i.e., when a CPU retrieves data before it is needed. Thus, this method requires hardware prefetching to be disabled.

The third approach allocates more memory than L1 cache can store, but less than than L2 cache can store. More specifically, twice the size of L1 cache were allocated in this experiment. In this approach, memory is accessed to fill the L1 cache twice. The first address written is then accessed again while measuring the number of cycles required to perform the read operation. In this way, the first address written is presumably evicted from L1 cache, and hence it is possible to measure L2 latency*. This approach consistently provided correct results and is for this reason used to produce the plots in this report.

* The reader should note that all cache levels are inclusive, i.e., an address evicted from L1 is still present in L2 since L2 is larger than L1.

```
void measure_l2()
{
   addr = allocate(2*L1_size);

   for(k = 0; k<N_MEASUREMENTS; k++)
   {
      for(j=0; j<2*L1_size; j++)
      {
         some_data = addr[j];
      }

      start_time = cycles::now();
      some_data = addr[0];
      stop_time = cycles::now();

      save_to_file(stop_time-start_time);
   }
}
```

Listing 4: Pseudo code of L2 measurements.

4.1.4 DRAM Measurements

In order to measure DRAM latency, it is necessary to fill all cache layers with data. The next step is to invalidate all addresses in the cache hierarchy by applying the *flush()* function [23]. This function is suggested by Intel, and makes sure that specified addresses are evicted from all caches and written back to DRAM if modified. Thus, the *flush()* function will clear all cache layers and hence force memory to be retrieved from DRAM the next time they are accessed. By measuring the number of cycles required to access allocated data after the cache hierarchy is flushed, it is possible to measure DRAM latency.

4.2 Virtual to Physical Address Translation

As described in Section 2.2, applications use virtual addresses. For this reason, it is necessary to define a method for translating each virtual addresses to a corresponding physical address. Since the page offset remains the same value for both physical and virtual addresses, the memory translation only requires the page frame number to which a given virtual address is mapped. In Linux,

this mapping is stored in a file named pagemap. Each process has its own pagemap file located in the following path: /proc/[process_id]/pagemap. Thus, it is possible to find the corresponding page frame number of a virtual address by searching the pagemap file for a given process ID. Listing 5 shows pseudo code for extracting the page frame number from a virtual address in the pagemap file. To better understand the pagemap structure, Appendix B provides example outputs of /proc/[process_id]/pagemap for different page sizes.

```
unsigned long get_page_frame_number_of_address(void *addr)
{
   //opening the pagemap file for the specific process
   pagemap = fopen("/proc/self/pagemap", "rb");
   //finding offset and moving the cursor to the proper position
   offset = (address / page_size) * length_for_each_page;
   page_frame_number = read(pagemap, offset);
   return page_frame_number;
}
```

Listing 5: Pseudo code for finding the page frame number.

As shown in Listing 6, this page frame number is used to translate a virtual address to a physical address.

```
uint64_t get_physical_address(char* addr)
{
   page_frame_number = get_page_frame_number_of_address(addr);
   offset = address % page_size;
   physical_address = combine(page_frame_number, offset);
}
```

Listing 6: Pseudo code of virtual to physical memory translation.

4.2.1 LLC Hash Function

To map a physical memory address to a LLC slice, a reverse engineered hash function from [1] is used. This paper describes three functions shown in Listing 7. The functions o_0, o_1, and o_2 accepts one 64-bit physical memory address and performs an XOR operation between certain bit positions to return either 1 or 0. The result from these three functions are combined to one integer value where o_0 represents bit 0 of this integer, o_1 bit 1, and o_2 bit 2. The value represents which

slice an address is mapped to. For example, if the return value (o_2, o_1, o_0) is (1, 0, 1), then this represents slice 5 since the binary sequence 101 is 5 in decimal notation.

$$O_0(b_{63},...,b_0) = b_6 \oplus b_{10} \oplus b_{12} \oplus b_{14} \oplus b_{16} \oplus b_{17} \oplus b_{18} \oplus b_{20} \oplus b_{22} \oplus b_{24} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{28} \oplus b_{30} \oplus b_{32} \oplus b_{33} \oplus b_{35} \oplus b_{36}$$

$$O_1(b_{63},...,b_0) = b_7 \oplus b_{11} \oplus b_{13} \oplus b_{15} \oplus b_{17} \oplus b_{19} \oplus b_{20} \oplus b_{21} \oplus b_{22} \oplus b_{23} \oplus b_{24} \oplus b_{26} \oplus b_{28} \oplus b_{29} \oplus b_{31} \oplus b_{33} \oplus b_{34} \oplus b_{35} \oplus b_{37}$$

$$O_2(b_{63},...,b_0) = b_8 \oplus b_{12} \oplus b_{13} \oplus b_{16} \oplus b_{19} \oplus b_{22} \oplus b_{23} \oplus b_{26} \oplus b_{27} \oplus b_{30} \oplus b_{31} \oplus b_{34} \oplus b_{35} \oplus b_{36} \oplus b_{37}$$

Listing 7: Hash function.

To represent these function in code, the bit positions are represented as hexadecimal numbers (there will be one hexadecimal number for each function). An AND operation is then performed between the physical address and each hexadecimal number. Finally, the parity of the results are combined to represent the slice number. As an example, the o_0 function contains the following bits: 6, 10, 12, 14, 16, 17, 18, 20, 22, 24, 25, 26, 27, 28, 30, 32, 33, 35, 36. A binary number where these bits are 1 (and the other bits are 0) can be represented as 0x1B5F575440 in hexadecimal notation.

4.2.2 Uncore Measurements

To measure the access time for a LLC slice a method is needed to determine which slice a memory address is mapped to. To find this mapping the hash function is used, but it only works on a few architectures and on a more modern CPU a different hash function must be is used. A more general method is to use uncore performance counters to find the mapping from memory address to LLC slice, and this method was used in the project to verify the implementation of the hash function. To use the uncore performance counters they have to be configured to measure a specific event which in this case is the LLC_LOOKUP event. This is done by writing specific values to certain control registers. Each LLC slice has a dedicated counter that is set up to count the number of LLC accesses. To perform the translation the following procedure is used:

1. Initialize the counter for every slice to monitor LLC_LOOKUP events.
2. Reset the counters to 0 and start the counting.

3. Access the physical memory address 10 000 times in order to make a clear distinction between the counters.

4. Read the counter for each LLC slice and the one with the highest value is where the physical address is mapped.

4.3 LLC Measurements

The approach to perform LLC measurements is slightly different from the previous measurements and exploits the fact that LLC is 16-way set associative compared to L1 and L2 which is 8-way set associative. For this reason, the complex addressing hash function was used to locate more than eight memory addresses that belong to the same LLC slice and same sets in L1, L2, and the LLC. Since L1 and L2 cache are 8-way set associative, only eight addresses can be located in the same set in at the same time, and hence the remaining addresses must be located in LLC. Thus, the LLC latency is measurable from the addresses that do not fit into L1 and L2. Listing 8 denotes the approach to perform LLC measurements.

```
void measure_llc_slice(slice)
{

   addr = get_addr_to_set_one(slice, 2*L2_associativity);

   for(k = 0; k<N_MEASUREMENTS; k++)
   {
      for(k = 0; k<2*L2_associativity; k++)
      {
         some_data = addr[k];
      }
      start_time = cycles::now();
      some_data = addr[0];
      stop_time = cycles::now();
      save_to_file(stop_time-start_time);
   }
}
```

Listing 8: Pseudo code of LLC measurements.

4.4 Multithreaded Slice-aware Allocation

Multithreaded slice-aware allocation exploits the fact that each CPU core has one LLC slice with minimal latency. Thus, it is possible to spawn separate threads for each core which only utilize on LLC memory with the lowest latency to that core. For example, if core 0 has the lowest latency to slice 0, then core zero will only use parts of memory which are mapped to slice 0. A similar approach applies to the rest of the CPU cores. The multithreaded slice-aware measurements were performed with different sized memory allocation in order to illustrate the performance gains from slice-aware allocation compared to normal random allocation. Listing 9 shows pseudo code of the multithreaded slice-aware allocation and random memory allocation.

```
num_threads = get_number_of_cpu_cores();
threads = create_thread_pool(num_threads);
for(auto& thread:threads)
{
   threads[thread].set_cpu_core(thread);
   threads[thread].measure(size_to_allocate);
   threads[thread].measure(size_to_allocate, random=true);
}
```

Listing 9: Pseudo code of slice-aware and random measurements.

The measurements performed from the threads shown in Listing 9 are partly shown in Listing 10. Each allocated cache line is accessed once following a uniform distribution. This measurement is repeated for N_MEASUREMENTS times which in this project is 10 000.

```
for N_MEASUREMENTS
{
    for num_cache_lines
    {
        index = get_uniform_index();
        some_data = *(set_addr[index]);
    }
}
```

Listing 10: Algorithm for accessing each allocated cache line.

4.5 The Library

A library written in C++ was developed during the course of the project to support and implement the various measurements performed. A link to the source code can be found in [4], and it can be compiled locally to perform the measurements. All measurements performed are based on read operations and not write operations to avoid potential problems with a write-back policy. Measurements supported includes L1 and L2 cache latency, DRAM latency, and LLC slice latency from every core. There is also support for comparing slice-aware memory allocation to normal memory allocation. It is also possible to find the mapping between a physical memory address and a LLC slice using uncore performance counters. Finally, the library contains various utility functions such as virtual to physical addresses translation, retrieval of cache sets, and a function to obtain LLC slice mappings using the hash function explained in Section 2.4.1.

The library is written using generic methods of measurements but adapted to work with the Intel Xeon E5-2667 v3 CPU. For this reason, the library will not function on any other processor without some minor modifications to the code. There are some assumptions about the system written into the code such as the availability of at least sixteen 1 GB hugepages, an Intel x86_64 architecture, that a cache line is 64 bytes, and that the system has 8 CPU cores.

The CPU Identification (CPUID) instruction is used by the library to detect a CPU's characteristics. Using CPUID, it is possible to detect various CPU related information such as CPU vendor, core count, and cache size [24]. CPUID utilizes EAX, EBX, ECX, and EDX assembly registers to fetch this information. The EAX register decides which information CPUID should retrieve, and hence the EAX register must be manually set prior to invoking CPUID to retrieve desired information. For example, if EAX is set to zero (0H) prior to invoking CPUID, then CPUID returns the CPU vendor in EBX, ECX, and EDX. Thus, in the case of an Intel CPU, EBX contains ”Genu”, EDX contains ”ineI”, and ECX contains ”ntel”, i.e., ”GenuineIntel”. A similar approach applies to retrieving other CPU related information as well.

Chapter 5

Results and Analysis

This chapter outlines the results from the measurement tool developed in Chapter 4. The access latency to different cache layers are evaluated, and the benefits from using slice-aware memory allocation compared to random memory allocation are discussed in detail. Finally, this chapter concludes the reliability of the acquired results.

5.1 Measurement Results

This section illustrates the access latency to different cache layers, DRAM, and the benefits from using slice-aware memory allocation compared to random memory allocation. The cache access latency is illustrated in box plot diagrams, while the slice-aware measurements are illustrated in a discrete line graph.

It is important to note that measuring the number of CPU cycles induce an overhead for the actual measurement. In our testbed, this overhead is 32 cycles. For this reason, 32 cycles are subtracted from all measurement results before they are illustrated. This is done to better illustrate the results and to facilitate a comparison with other papers.

5.1.1 L1, L2, and DRAM Results

In this subsection, the results for the L1, L2, and DRAM measurements are outlined. The measurements provide the latency in terms of cycles for accessing L1, L2, and DRAM. Figure 5.1 illustrates the access latency for L1 and L2 cache. L1 median latency measures 4 cycles, while L2 has a median of 12 cycles. Thus, 4 cycles were required to access L1 memory and 12 cycles where required to access L2 memory. In the case of L1 and L2 measurements, the upper quartile of the box

plot overlap the median lines, this means that the data is heavily skewed to the left. This behaviour is expected* when measuring computer performance since every measurement should yield the same execution time. For this reason, the median line is made thicker in Figure 5.1 to better indicate the median placement. In some cases, the whiskers of the box plots overlap with the edge of the boxes.

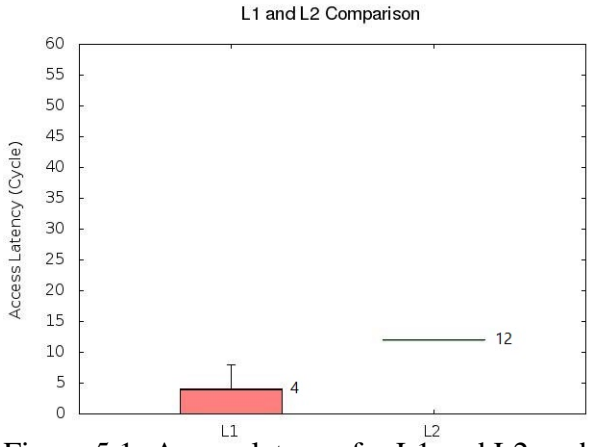

Figure 5.1: Access latency for L1 and L2 cache.

Figure 5.2 depicts the number of cycles required to access L1, L2, and DRAM. The median DRAM latency was 349 cycles. The DRAM latency is much higher than L1 and L2 which is clearly visible in Figure 5.2.

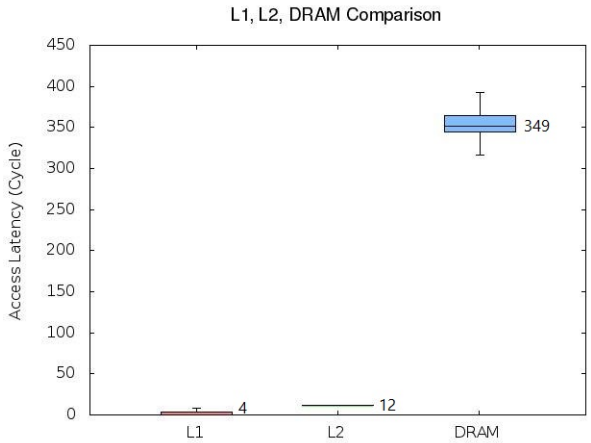

Figure 5.2: Access latency for L1, L2, and DRAM.

* Expected because the measurements are performed on an isolated CPU.

5.1.2 LLC Results

Figure 5.3 illustrates the time in cycles for core 0 to access different LLC slices. Core 0 has lowest latency to slice 0 and highest latency to slice 1. The differences in latency occurs due to the physical placement of cores and LLC slices on the CPU die. As seen in Figure 1.1, CPU cores and LLC slices are connected via a ring bus, hence the distance from one core to each LLC slice varies with a core's placement. Thus, the access time depends on the distance from one core to a LLC slice. Similar results are observed in Figure 5.4 where the access times for core 1 to different slices are illustrated.

The LLC access time for core 1 varies from core 0 since core 1 is located on a different part of the ring bus. It can be seen that core 1 has minimal latency to slice 1 and maximal latency to slice 0, while core 0 has minimal latency to slice 0 and maximal latency to slice 1. The results for the other CPU cores are available in Appendix A and shows similar results, i.e, every core has minimal latency to its own* slice. The purpose of this project is to exploit this knowledge and to force each CPU core to only utilize a LLC slice with minimal latency.

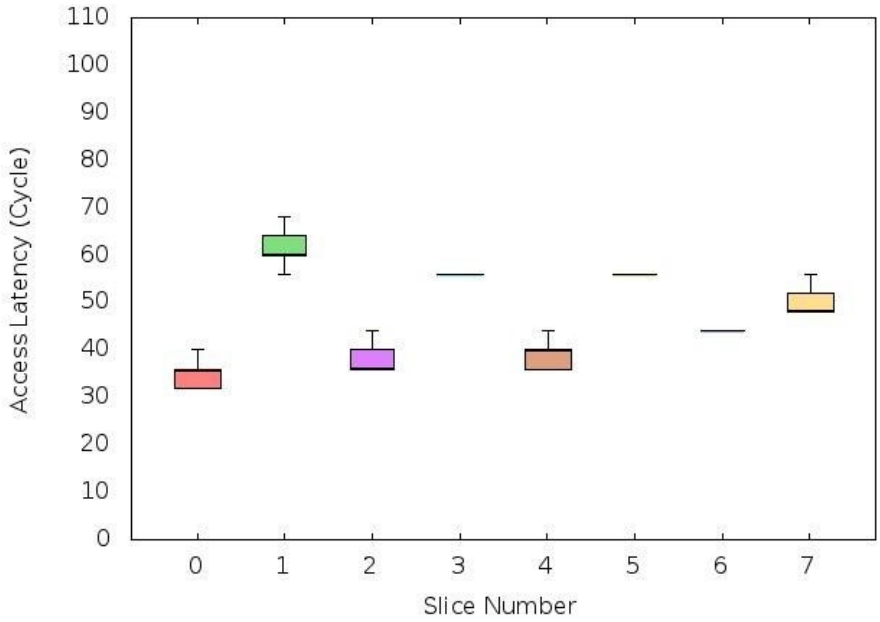

Figure 5.3: LLC latency from core 0.

* The slice number matches the core number.

In this way, CPU performance should improve when all cores use a LLC slice with minimal latency. This paper distinguishes between slice-aware allocation and random memory allocation. In slice-aware allocation, each CPU core is utilizing LLC memory with minimal latency, while in random memory allocation, each LLC slice is equally likely to be used, i.e., random memory allocation is independent on the processing CPU core (the core which is executing an application). For this reason, it is expected that the CPU can execute more Operations Per Second (OPS) when slice-aware memory allocation is applied.

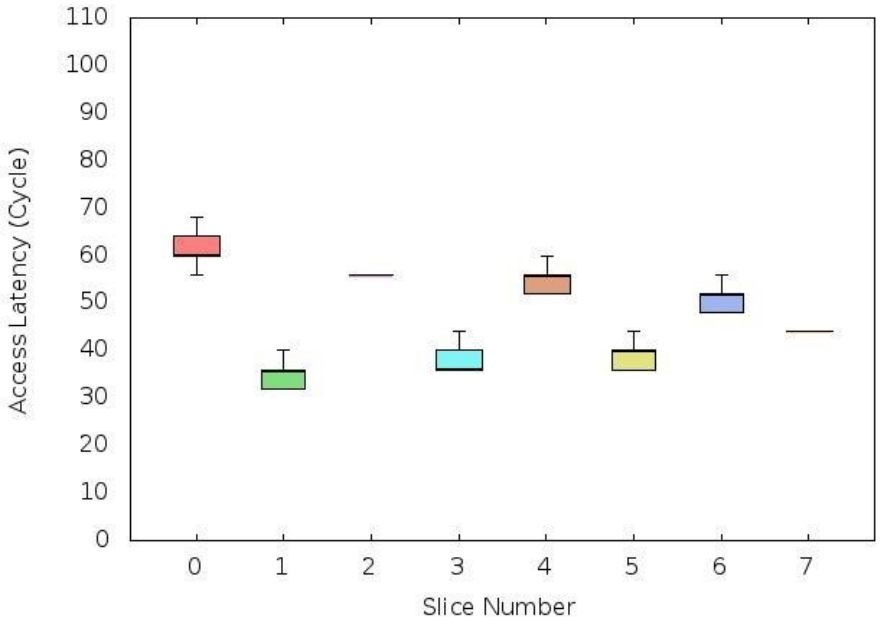

Figure 5.4: LLC latency from core 1.

5.1.3 Slice-aware vs Random Allocation

Figure 5.5 illustrates the benefit of using slice-aware memory allocation compared to random memory allocation. In this measurement, one thread is running on each core, hence the allocated size in Figure 5.5 is in bytes per thread. Table 5.1 shows corresponding statistical data to Figure 5.5. The results are equal for the two cases when the allocated memory is $\leq 2^{17}$ bytes. This is expected as L1 and L2 caches are unique to each core; hence slice-aware memory allocation has no impact on these measurements, and for this reason they will perform similar. The advantage of using slice-aware memory allocation is noticeable between 2^{17} and 2^{22} bytes with an improvement of more than 5% at 2^{21} bytes.

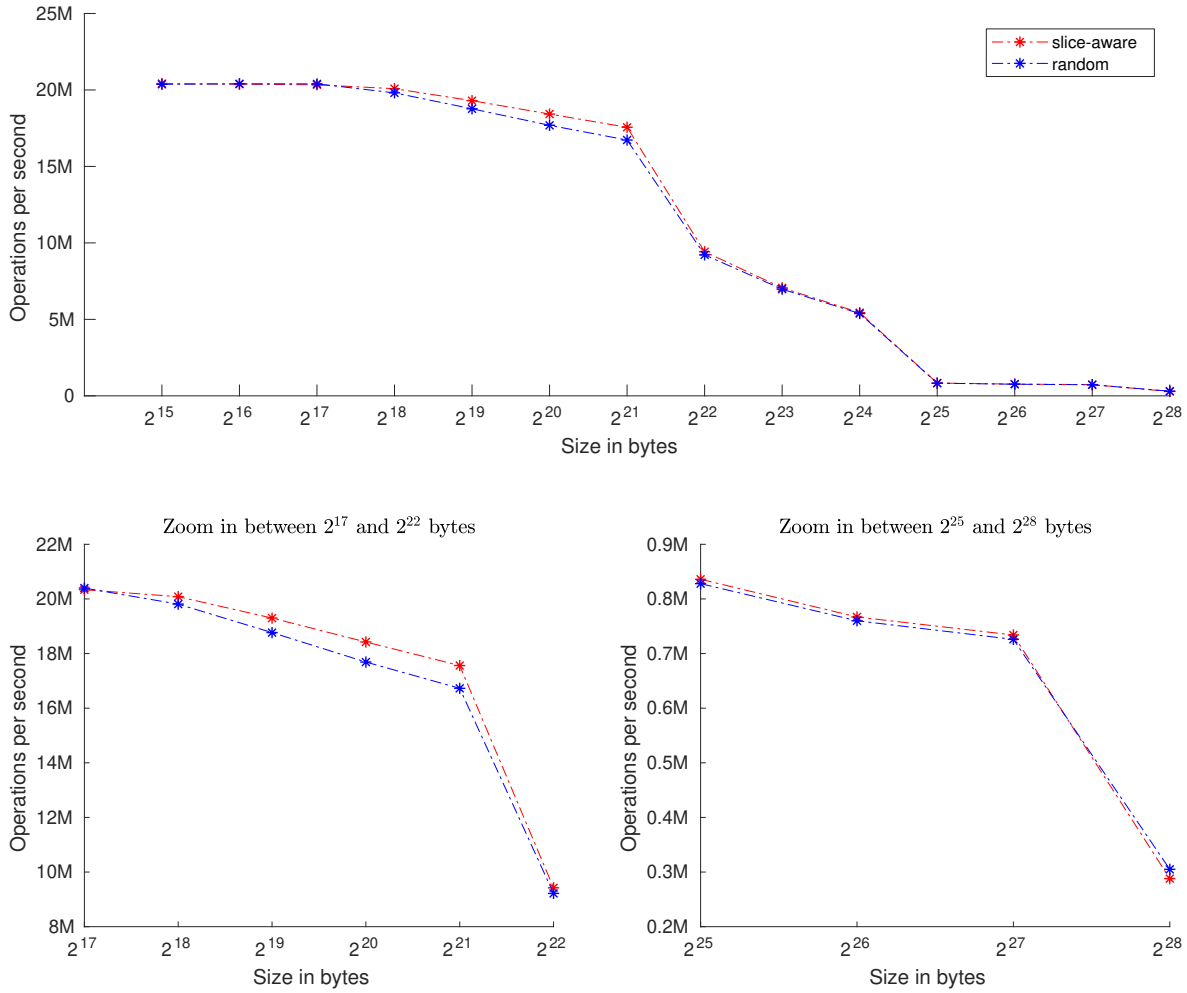

Figure 5.5: Slice-aware vs random memory allocation (read operations).

Table 5.1: Statistical data corresponding to Figure 5.5.

Size (Bytes)	Size	Slice-aware (MOPS)	Random (MOPS)
2^{15}	32 KiB	20.4	20.4
2^{16}	64 KiB	20.4	20.4
2^{17}	128 KiB	20.3	20.4
2^{18}	256 KiB	20.1	19.8
2^{19}	512 KiB	19.3	18.8
2^{20}	1 MiB	18.4	17.7
2^{21}	2 MiB	17.6	16.7
2^{22}	4 MiB	9.4	9.21
2^{23}	8 MiB	7.1	7.0
2^{24}	16 MiB	5.4	5.4
2^{25}	32 MiB	0.84	0.83
2^{26}	64 MiB	0.77	0.76
2^{27}	128 MiB	0.73	0.73
2^{28}	256 MiB	0.29	0.31

However, it is not until 2^{28} bytes that random memory performs is better, but there is a steady decrease from 2^{22} bytes. Slice-aware memory allocation is greatly beneficial when the data size fits in an LLC slice. As the allocated size increases, the advantage of slice-aware memory allocation is reduced since there are more cache misses when trying to fit a large chunk of data in one LLC slice. The steep decline in OPS between 2^{21} and 2^{22} bytes occurs since the allocated data exceeds the memory capabilities of a LLC slice. The second decline in OPS, between 2^{24} and 2^{25} bytes, occurs when the allocated size is larger than the whole LLC memory. Thus, the measurement algorithm in Listing 10 will access cache lines that must be retrieved from DRAM; hence the OPS will drastically decrease.

5.2 Reliability Analysis

The L1, L2, LLC, and DRAM measurements were repeated 10 000 times per test to reduce the impact of occasional noise. Executing the tests multiple times has also shown that the median does not change between measurements. For the uncore measurements, we also chose to poll each address 10 000 times to have a clear distinction between the counter for the mapped slice and the other counters. It might have worked with a lower poll count but polling 10 000 times has proved to always give correct unambiguous results while still finishing within a reasonable time. The comparison between slice-aware memory allocation and random allocation is the average number of operations per millisecond calculated from 8* measurements.

5.3 Validity Analysis

This section compares the result found in this project with previous findings from multiple known sources. The results are expected to closely match the reference values. Table 5.2 shows results obtained in this project and reference values from external sources. There are 4 CPU cycles required to access L1 cache, 12 cycles for L2 cache, 36 cycles for LLC, and 349 cycles for DRAM.

Table 5.2: Results from this project (median in cycles).

	L1	L2	LLC	DRAM
(Latency in cycles)	4	12	36†	349

Table 5.3 denotes the measurement results for different sources that are used as references throughout this paper. Yarom et al [10] got 4 cycles for L1 and 7 cycles for L2. Drepper et al [25] presented 3 cycles for L1, 14 cycles for L2, and 240 cycles for DRAM. It is important to note that these measurements are performed on different Intel CPU Stock Keeping Units (SKUs) and hardware configurations.

* The tests are performed on 8 threads running in parallel on 8 different cores. † Median access time for core 0 to slice 0.

Table 5.3: Results from sources (median in cycles).

L1	L2	LLC	DRAM	Source
4	12	26-31*	-	[12]
4	7	26-31	-	[10]
3	14	-	240	[25]

A comparison between Table 5.2 and Table 5.3 shows that the results obtained in this project are reasonably close to the external sources. The expected L1 latency is 3-4 cycles which is similar to the value obtained in this project. This also applies for L2 latency which is in the range of 7-14 cycles. The obtained results for LLC and DRAM latency are a bit higher than the reference values. This is believed to be due to different hardware in the testbeds.

5.4 CPU Isolation

As explained in Section 4.1.1, one CPU socket was isolated from all user space processes. Isolating the entire socket also reserved the entire LLC. CPU isolation was performed in order to get more consistent and accurate measurements. The same measurements could be performed without CPU socket isolation, but with reduced accuracy because of potential interference from neighboring processes.

* Latency of LLC varies with product segment and SKU[12].

Chapter 6

Conclusions and Future Work

This chapter concludes this project and describes the future work to be carried out. Section 6.1 provides the project's conclusions and briefly describes its outcomes. Section 6.2 presents suggestions for future work. In Section 6.3, the impact of the solution is examined from both an economic and environmental perspective.

6.1 Conclusion

Due to rapid technological advancements within the hardware industry, applications become faster and can deliver a better user experience. Unfortunately, many applications can not deliver best possible performance since software developers do not utilize on slice-aware memory allocation.

This project was based on the hypothesis that awareness of the underlying memory architecture in programming can improve the performance of applications. At the time when this project started, there was no available tool or library (known to the authors) to prove this hypothesis. Therefore, a C++ library was developed for this purpose. This library provide measurements with two different memory allocation schemes: random allocation and slice-aware allocation. The results show that slice-aware memory allocation can greatly enhance (more than 5% improvement) the performance of applications that mainly utilize on LLC memory. However, the performance enhancement is mainly noticeable when the allocated memory is larger than the size of L2 cache and smaller than the size of a LLC slice. In all other cases, the performance of slice-aware allocation is similar or worse than random memory allocation. This proves that minor improvements in memory management can greatly enhance the performance of applications when an application mainly utilize LLC memory. Thus, the hypothesis is true, i.e., knowledge of the underlying memory architecture can improve application

performance.

However, slice-aware memory allocation can perform worse than random allocation if the allocated memory exceeds the LLC. This is believed to be the case since slice-aware memory allocation suffers from an increased cache line eviction probability when a large block of memory is allocated in a single slice. Thus, random memory allocation has a lower cache line eviction probability since random allocation can make use of the entire LLC.

In this project the main goals were met. The memory organization of the Intel architecture in a Linux-based operating system were deeply studied. An architecture-aware library was developed which can measure access times to different levels of the memory hierarchy. This library can also perform LLC measurements using a slice-aware allocation scheme. It was also proved that slice-aware allocation can improve the efficiency of an application. However, as already mentioned, this slice-aware allocation technique is not preferred in all cases.

As illustrated in Figure 5.5, the maximum value in OPS is close to 20 million. While this value could be higher depending on how the measurements are performed, what is important is how the results from random allocation differ from slice-aware allocation, and not specifically how many operations per millisecond they perform.

6.2 Future Work

This section provides multiple topics for future work:

- The CPU cores were isolated during the experiments for slice-aware memory allocation. A possible avenue to explore is thus the consequence of having a noisy neighbor impact the measurements. The experiments can be done in three different phases: (1) the application and a noisy neighbor utilize the same CPU core, (2) the application utilizes a dedicated core while the noisy neighbor is shared among all other cores, and (3) both the application and the noisy neighbor are shared among all cores.

- The measurements in this project were only performed in user space. Therefore, kernel space measurements could be performed to verify the accuracy of user space measurements.

- This project only focused on Intel's Haswell architecture. For this reason,

the library could be extended to support other architectures as well as other CPU vendors.

- The library only support Linux OS in its current implementation. Therefore, one opportunity is to develop support for Windows and Mac OS. In this way, the library can be used on other operating systems as well.
- In its current state, the library can only measure L1, L2, LLC, and DRAM. Thus, it can be extended to measure access times to Solid State Drive (SSD) and Hard Disk Drive (HDD) storage.

6.3 Reflections

With the great advancements in modern technology, today's applications are expected to be very fast. In order to satisfy the consumers' requirements, the suppliers must produce products that consume less energy but with better performance. The solution provided in this project has the potential to improve the performance of applications in several cases without changing hardware or resources. Implementing the solution could lead to a cost reduction for the supplier, as well as a decrease in power consumption; thus reducing the emission of greenhouse gases.

Bibliography

[1] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters," in *Research in Attacks, Intrusions, and Defenses*, ser. Lecture Notes in Computer Science, H. Bos, F. Monrose, and G. Blanc, Eds. Springer International Publishing, 2015. ISBN 978-3-319-26362-5 pp. 48–65.

[2] Karl Rupp, "42 Years of Microprocessor Trend Data," Feb 2018, Accessed 2019-01-04. [Online]. Available: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

[3] Carlos Carvalho, "The Gap between Processor and Memory Speeds." Department of Information Technology University of Minho, 2002, p. 8. [Online]. Available: https://pdfs.semanticscholar.org/6ebe/c8701893a6770eb0e19a0d4a732852c86256.pdf

[4] J. Fjällid, N. Eneroth, H. Bjørseth, F. Zareafifi, and A. Shokrollahi, "MEM/ik2200-mem," GitHub Respository. [Online]. Available: https://gits-15.sys.kth.se/MEM/ik2200-mem

[5] Anne Håkansson, "Portal of Research Methods and Methodologies for Research Projects and Degree Projects." Las Vegas USA: CSREA Press U.S.A, 2013. ISBN 1-60132-243-7 pp. 67–73. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-136960

[6] David Black-Schaffer, "Virtual Memory: 2 Three problems with Memory," Jul 2014, YouTube video, Accessed: 2019-01-03. [Online]. Available: https://www.youtube.com/watch?v=eSPFB-xF5iM

[7] S. Haldar and A. Aravind, *Operating Systems*. Pearson Education India, 2010. ISBN 978-81-317-3022-5

[8] Andrew S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 978-0-13-600663-3

[9] N. Weizer and G. Oppenheimer, "Virtual Memory Management in a Paging Environment," in *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, ser. AFIPS '69 (Spring). New York, NY, USA: ACM, 1969. doi: 10.1145/1476793.1476834 pp. 249–256. [Online]. Available: http://doi.acm.org/10.1145/1476793.1476834

[10] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel Last-Level Cache," *IACR Cryptology ePrint Archive*, p. 905, 2015. [Online]. Available: https://pdfs.semanticscholar.org/8aa0/81092383f8622a77fe25e1da2ab81df2438b.pdf?_ga=2.263737225.406428703.1546497338-547835323.1546497338

[11] Timothy Prickett Morgan, "Drilling Down Into The Xeon Skylake Architecture," Aug. 2017, Accessed: 2019-01-03. [Online]. Available: https://www.nextplatform.com/2017/08/04/drilling-xeon-skylake-architecture/

[12] Intel Corporation, "Intel® 64 and IA-32 Architectures Optimization Reference Manual," 2016, Manual. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[13] Chuanjun Zhang, "An efficient direct mapped instruction cache for application-specific embedded systems," in *Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*, Jersey City, NJ, USA, Sep. 2005. doi: 10.1145/1084834.1084850 pp. 45–50.

[14] Stony Brook University, "CSE502: Computer Architecture," p. 23, University Course, Accessed: 2019-01-05. [Online]. Available: https://compas.cs.stonybrook.edu/course/cse502-s14/lectures/cse502-L3-memory-hierarchy-and-caches.pdf

[15] Aditya Bhutra, "Reviewing various Cache Replacement Policies," 2015. [Online]. Available: https://www.researchgate.net/publication/301546620_Reviewing_various_Cache_Replacement_Policies

[16] Onur Aciiçmez, "Yet another MicroArchitectural Attack:: exploiting I-Cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture - CSAW '07*. Fairfax, Virginia, USA: ACM Press, 2007. doi: 10.1145/1314466.1314469. ISBN 978-1-59593-890-9 p. 11. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1314466.1314469

[17] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Topics in Cryptology – CT-RSA*

2006. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 3860, pp. 1–20. ISBN 978-3-540-31033-4 978-3-540-32648-9. [Online]. Available: http://link.springer.com/10.1007/11605805_1

[18] Intel Corporation, "Intel® Xeon® Processor E5 and E7 v3 Family Uncore Performance Monitoring Reference Manual," Jun. 2015, Manual. [Online]. Available: https://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-v3-uncore-performance-monitoring.html

[19] Red Hat, "3.13. Isolating CPUs Using tuned-profiles-realtime - Red Hat Customer Portal," Accessed: 2019-01-05. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/isolating_cpus_using_tuned-profiles-realtime

[20] Red Hat, "6.3. Configuration Suggestions - Red Hat Customer Portal," Accessed: 2019-01-05. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide/sect-red_hat_enterprise_linux-performance_tuning_guide-cpu-configuration_suggestions

[21] L. McVoy and C. Staelin, "LMbench - Tools for Performance Analysis," Accessed: 2019-01-06. [Online]. Available: http://www.bitmover.com/lmbench/

[22] Gabriele Paoloni, "How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures," Sep. 2010, White Paper. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf

[23] Intel Corporation, "User space software for Intel(R) Resource Director Technology: intel/intel-cmt-cat," Accessed: 2019-01-02. [Online]. Available: https://github.com/intel/intel-cmt-cat/blob/master/examples/c/PSEUDO_LOCK/dlock.c#L67-L87

[24] "CPUID:x86 Instruction Set Reference," Accessed: 2019-01-06. [Online]. Available: https://c9x.me/x86/html/file_module_x86_id_45.html

[25] Ulrich Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11, 2007, Accessed 2019-01-04. [Online]. Available: http://cs.curs.pub.ro/wiki/asc/_media/asc:lab5:what_every_programmer_should_know_about_memory_by_ulrich_drepper_.pdf

Appendix A

LLC access latency from different cores

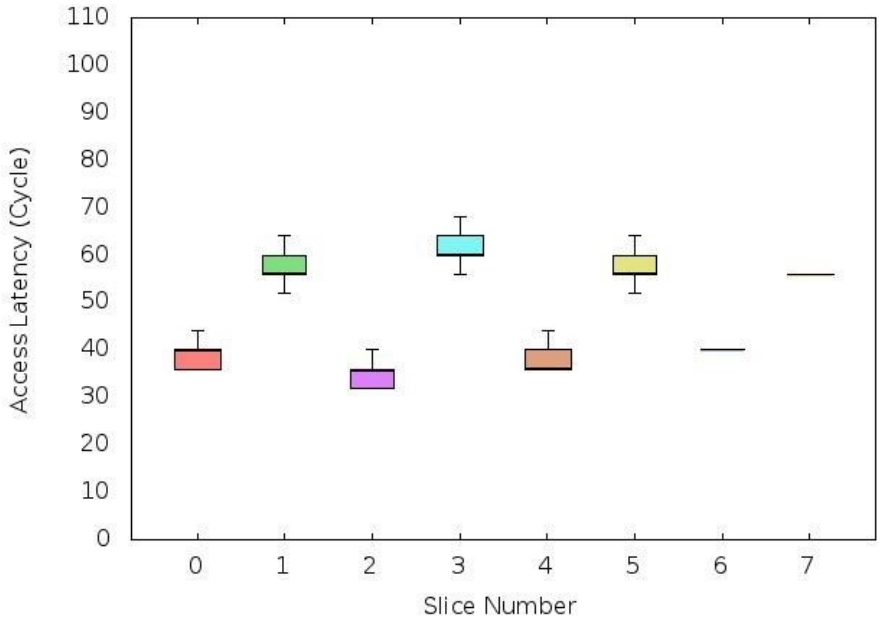

Figure A.1: LLC latency from core 2.

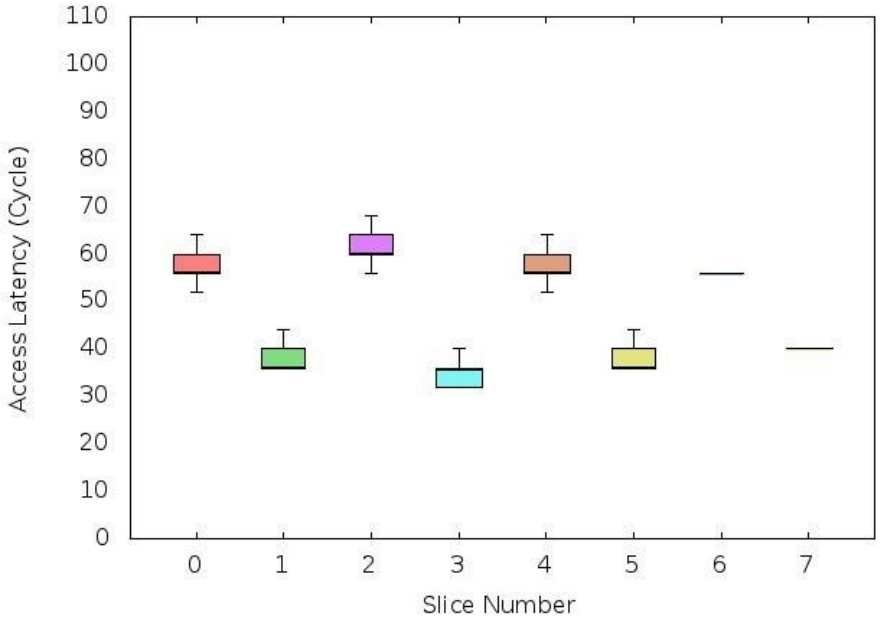

Figure A.2: LLC latency from core 3.

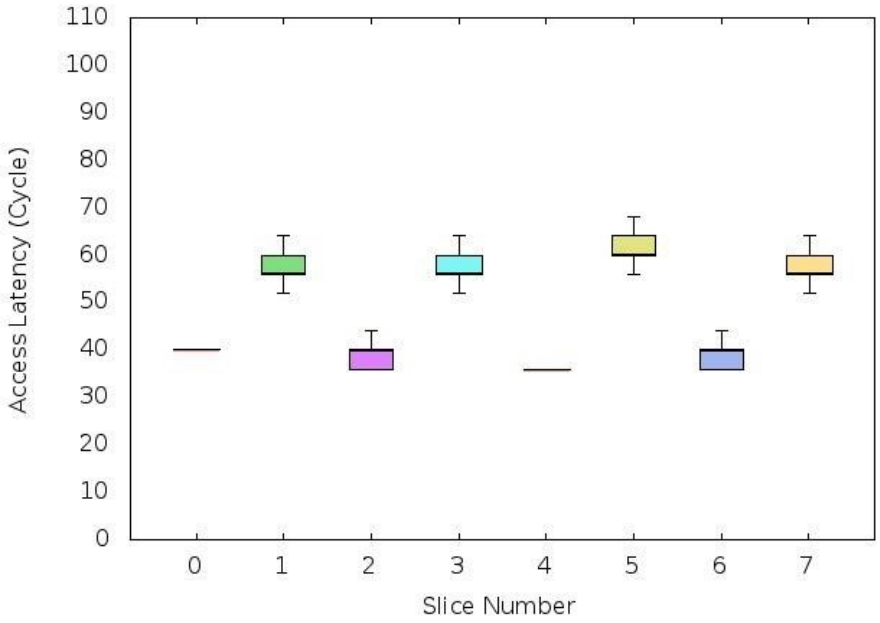

Figure A.3: LLC latency from core 4.

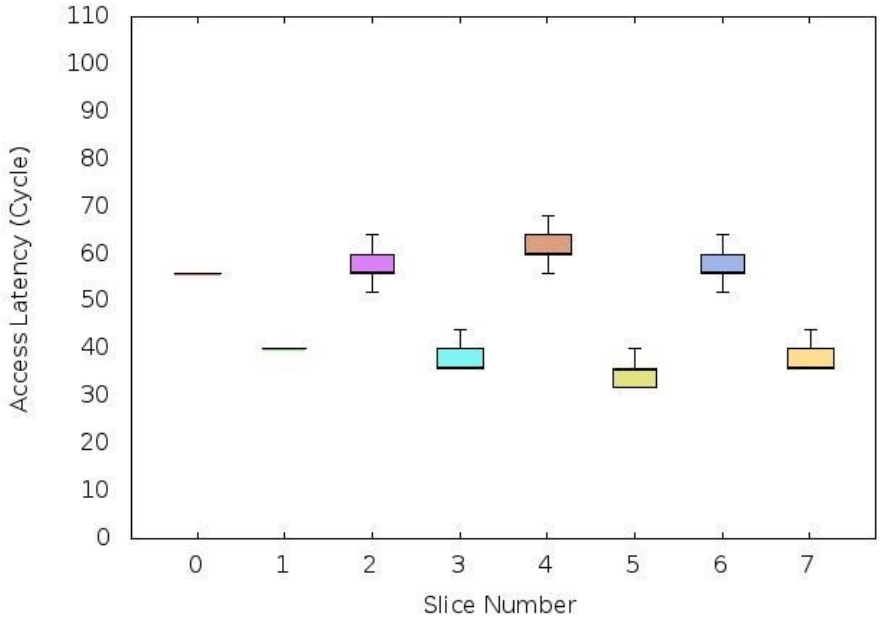

Figure A.4: LLC latency from core 5.

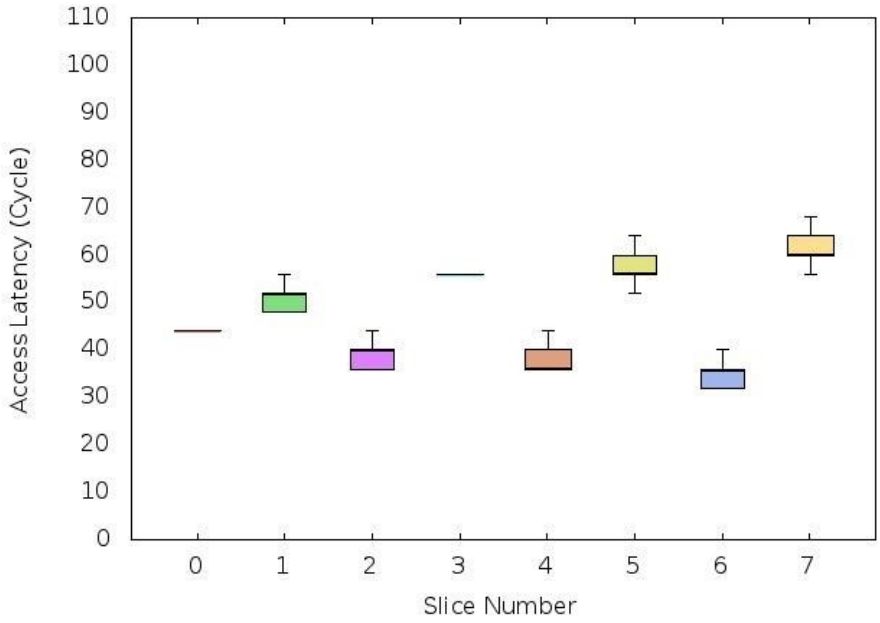

Figure A.5: LLC latency from core 6.

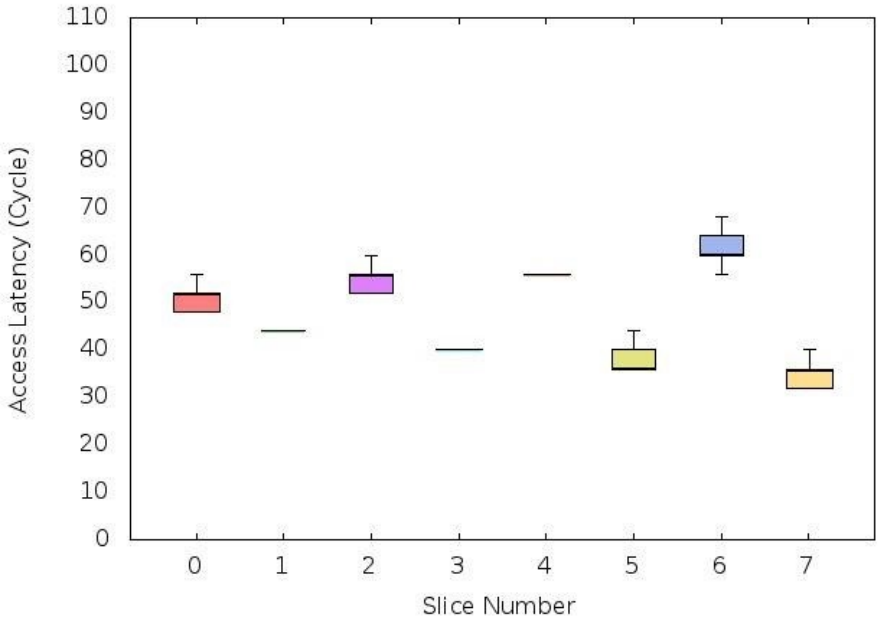

Figure A.6: LLC latency from core 7.

Appendix B

Pagemap Process in Linux

Figure B.1 and Figure B.2 illustrate the content of the pagemap file in the process of finding a page frame number for a virtual address in Linux OS. In Figure B.1, the page size is 4 KB, while in Figure B.2, the page size is 1 GB.

```
*****************************************************************************
Virtual Address: 0x7ffd9f80bd80
The page offset is:

(address_of_page/each_page_size)*data_for_each_page
                            ==> ( 0x7ffd9f80bd80 / 4096 )* 8 = 274218729728
Inside of /proc/self/pagemap:

   7fb16b820  --->  8180000001d29ea2  <---- Entry for the first page of the allocation
   7fb16b821  --->  8180000001d1c1fb
   7fb16b822  --->  8180000001d1e2a3
   7fb16b823  --->  8180000001d1e16d
   7fb16b824  --->  8180000001d1dc7f
   7fb16b825  --->  8180000001d1d82b
   7fb16b826  --->  8180000001d1dea0
   7fb16b827  --->  8180000001d242f1
   7fb16b828  --->  8180000001d1c494
   7fb16b829  --->  8180000001d22494
   7fb16b82a  --->  8180000001d1dbee
   7fb16b82b  --->  8180000001d226cf
   7fb16b82c  --->  8180000001d1de9c
   7fb16b82d  --->  8180000001d1df5c
   7fb16b82e  --->  8180000001d1deea
   7fb16b82f  --->  8180000001d42d4f
   7fb16b830  --->  8180000001d1d581
   7fb16b831  --->  8180000001d1cc3e
   7fb16b832  --->  8180000001d1ed55
   7fb16b833  --->  8180000001d1db7f
   7fb16b834  --->  8180000001d1dca6

==> The page frame number corresponds to the 55 lowest significant bits:  1d29ea2
```

Figure B.1: Contents of /proc/self/pagemap for 4 KB page size.

```
********************************************************************************
Virtual Address: 0x7ffe91b62a80
The page offset is:
 (address_of_page/each_page_size)*data_for_each_page
                              ==>  ( 0x7ffe91b62a80 / 4096 ) * 8 = 273743347712
Inside of /proc/self/pagemap:

   7f78c0000  --->  8180000001ec0000  <---- Entry for the first page of the allocation
   7f78c0001  --->  8180000001ec0001
   7f78c0002  --->  8180000001ec0002
   7f78c0003  --->  8180000001ec0003
   7f78c0004  --->  8180000001ec0004
   7f78c0005  --->  8180000001ec0005
   7f78c0006  --->  8180000001ec0006
   7f78c0007  --->  8180000001ec0007
   7f78c0008  --->  8180000001ec0008
   7f78c0009  --->  8180000001ec0009
   7f78c000a  --->  8180000001ec000a
   7f78c000b  --->  8180000001ec000b
   7f78c000c  --->  8180000001ec000c
   7f78c000d  --->  8180000001ec000d
   7f78c000e  --->  8180000001ec000e
   7f78c000f  --->  8180000001ec000f
   7f78c0010  --->  8180000001ec0010
   7f78c0011  --->  8180000001ec0011
   7f78c0012  --->  8180000001ec0012
   7f78c0013  --->  8180000001ec0013
   7f78c0014  --->  8180000001ec0014

 ==> The page frame number corresponds to the 55 lowest significant bits:  1ec0000
```

Figure B.2: Contents of /proc/self/pagemap for 1 GB page size.