Karlsruhe Institute of Technology

An Analysis and Software Reimplementation of Intel Optane DC Memory Mode

Bachelor's Thesis
submitted by

cand. inform. Michel Max

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Lukas Werling, M.Sc.

24. January 2020 – 28. June 2020

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, June 28, 2020

Abstract

Intel Optane DC Persistent Memory provides high density, byte-addressable main memory with larger capacity and slightly lower bandwidth and speed than DRAM. To utilize this larger quantity of slower main memory easily, Intel's hardware provides the so-called Memory Mode, which uses DRAM as a cache for the persistent memory. [23] [19]
This work aims to determine the cache organization of the Memory Mode, by designing and using a series of benchmarks, and aims to reimplement a software variant of the Memory Mode through a user space library to provide a more flexible and transparent alternative to the Memory Mode.

Contents

Chapter 1

Introduction

Intel and Micron have recently released a new type of non-volatile main memory called 3D XPoint (read CrossPoint). The underlying technology seems to be phase change memory (PCM) [24]. With this PCM technology, Intel and Micron can produce non-volatile, byte-addressable main memory. Compared to Dynamic Random Access Memory (DRAM), it is cheaper and denser, but a bit slower. More specifically, it has higher read latency, similar write latency, and much lower write bandwidth. This can be seen in Table 1.1 and Figure 1.1 respectively. Intel markets this new technology under the name *Intel Optane DC Persistent Memory* [9]. As this new memory is cheaper and denser than DRAM, it allows us to get larger amounts of main memory than before. This can be especially useful for in-memory databases, which need enormous amounts of main memory (as the entire data resides in main memory).

For the remainder of this work, *Intel Optane DC Persistent Memory* shall be called NVM, short for *non-volatile memory*.

Intel's NVM is designed to work together with DRAM in a system. It has two main operating modes: *App Direct Mode* and *Memory Mode*. In App Direct Mode, both the NVM and the DRAM are visible to the processes running on the system. The DRAM is the main memory, while the NVM is byte-addressable storage in this configuration. In Memory Mode on the other hand, both NVM and DRAM together form the main memory. The DRAM is a transparent cache for the larger and slower NVM. The cache is managed by the memory controller, and the total capacity is equal to the capacity of the NVM. Both of these modes are explained in much more detail in the background Section 2.1. [10]

Intel provides this Memory Mode to allow programs to immediately start using the new NVM without any adaptation. Nevertheless, those programs could increase their performance if they were adapted to better work with the specific cache organization the Memory Mode has. Since Intel has so far not released how the cache works internally, we tried to build a series of benchmarks to experimen-

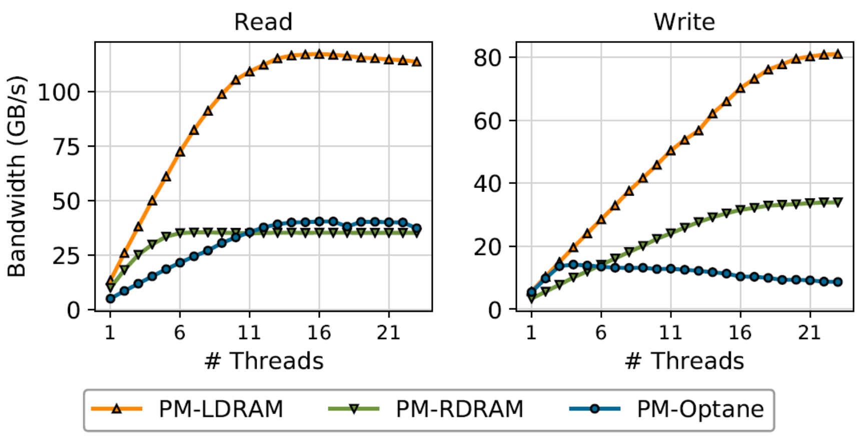


Figure 1.1: Comparison of read and write bandwidths between local and remote DRAM, and NVDIMMs. [23]

tally determine the information ourselves. This was met with mixed success. We also tried to develop our own software implementation of the Memory Mode, that is more transparent and adaptable than the Memory Mode. Our software implementation is still work in progress, and is not yet ready to replace the Memory Mode, although results show that it could at some point in the future. This could provide an alternative to software developers who do not wish to adapt their software to use the App Direct Mode, but would still like to adapt their program to the Memory Mode.

Technology	Random Read Latency	Random Write Latency	Largest DIMM available	Price of 128 GB DIMM
DRAM	81 ns	86 ns	128 GB	1.350$
NVDIMM	305 ns	94 ns	512 GB	850$

Table 1.1: A comparison of the volatile DRAM and the non-volatile PCM memory technologies. Write latencies are close, but the real difference is the write bandwidth as can be seen in Figure 1.1 Latency from [23], availability and prices from [19] and [1]

Chapter 2

Background

This sections explains some of the technology and methods used in this work.

2.1 Intel Optane DC Persistent Memory (NVM)

Intel Optane DC Persistent Memory is a new type of non-volatile main memory developed by Intel and Micron. This means it is byte-addressable, and data will not be lost if the power is cut. This is in contrast to DRAM, which needs a steady supply of power to retain any data [21]. Since this NVM is denser and cheaper than DRAM, but also a bit slower as shown in Table 1.1, it can be placed between DRAM and traditional persistent storage, such as SSDs and HDDs, on the memory hierarchy [15]. The NVM can be used in two main modes of operation, the Memory Mode and the App Direct Mode. The next two sections will describe these two modes.

2.1.1 App Direct Mode

In this mode, the NVM and the DRAM are visible in the system. The DRAM is used as standard main memory. The NVM on the other hand can be set up in different ways. The NVM is divided into regions according to the physical layout of the memory. One region is generally a physically contiguous span of NVM [17], such as the NVM belonging to one NUMA node. This NVM can then be further divided into namespaces, which can operate in four different modes [17]:

fsdax This option creates a block device out of the namespace with the DAX (see section2.1.3) file system capability. This device is then accessible in the `/dev/` directory, and a file system can be created on it. This is the default.

devdax This mode provides a single character device file, also under `/dev/`.

sector This is for legacy file systems and small boot devices.

raw This provides a simple memory disk without DAX capability.

During the reimplementation part of this work in Section 5, we use the `fsdax` mode.

2.1.2 Memory Mode

In this mode, the NVM and DRAM are no longer visible as individual entities, but both together form the main memory of the system. The DRAM is used as a cache to the slower and larger NVM. The main memory capacity is equal to the amount of NVM installed on the system, making the DRAM completely transparent. The memory controller takes care of the cache, so we do not know how this cache works, which is one of the main subjects of this work. [10]

This mode is a lot easier to use than the App Direct mode, since you only need to set the system to Memory Mode, and then you can use it the same way as any other system without NVM.

2.1.3 Direct Access File Systems (DAX)

Direct Access (DAX) is a feature that has been added recently to some file systems, like `ext4` and `xfs`, to make better use of non-volatile main memory. Traditionally under Linux, reads and writes on open files are buffered in the page cache in DRAM. When a process calls `mmap` to map parts of a file into its virtual address space, the file is copied into the same page cache and that copy is then mapped into the process's virtual address space. When using a file system that supports DAX and a file that is stored on non-volatile main memory, we do not need to copy the file into the page cache. Instead, we can map the non-volatile main memory directly into the process's virtual address space. We can then read and write directly on the NVM, so we save the copy of the file into the page cache. [4]

Chapter 3

Related Work

Since most of the work presented in this section only covers one part of this work, this section is divided into two parts. Each work is in the section it relates most to.

3.1 Analysis

Researchers at the Non-Volatile Systems Laboratory at the University of California, San Diego have published a paper extensively analyzing the Memory Mode, the App Direct mode, and applications designed for non-volatile main memory. While they claim to know how the Memory Mode works, they are missing some sources. This is one of the earlier works analyzing the Memory Mode. [23].

X-Mem is a tool to measure the performance of main memory. It is described and used in a paper by its authors, Mark Gottscho et al. In the paper, the tool is used to infer CPU cache organization of cloud provider's systems, where the exact hardware is unknown. This is similar to our goal, where we are trying to gather information about an unknown cache organization, although the Memory Mode's DRAM cache is much larger and necessitate different techniques [22].

Andreas Abel wrote a master's thesis in which they explore methods to determine the CPU cache organization experimentally, most notably the cache size and associativity. We attempt to determine those characteristics in the Memory Mode. Their methods methods seem quite similar to the ones we use to determine those characteristics, but differ a bit since they are designed for small CPU caches, instead of the large cache of the Memory Mode. [18].

3.2 Reimplementation

Xiangyao Yu et al. developed Banshee, a DRAM cache between in- and off-package DRAM. While they are not using persistent memory, they are still implementing a cache in DRAM, which we will also be doing in Section 5. The techniques described in their publication seem to be a bit more advanced than the techniques used in our reimplementation, but they could be used if our reimplementation is further developed in the future [26].

Chapter 4

Analysis

In this chapter, we will discuss the design, implementation and results of a series of benchmarks designed to shed light on the inner workings of the Memory Mode.

4.1 Benchmark Design

In this section, we will talk about the design of the benchmarks. The goal of these benchmarks is to determine the characteristics of the Memory Mode (e.g., cache line size and prefetching behavior). These properties of the Memory Mode help us better understand it, and implement our own version of the Memory Mode in Section 5.

4.1.1 Cache Size Test

The goal of this test is to determine the actual size of the DRAM cache. Of course, the answer could be that the size of the DRAM cache is simply the amount of DRAM installed on the system. It is also possible that the Memory Mode uses some internal data structures to manage the cache, and that those data structures are stored in the DRAM. This would mean that some part of the DRAM is not available as cache, but is instead reserved.

Since our system is divided into two NUMA nodes, with each node having half of the DRAM and NVM respectively, this test can also check if pinning a process to the memory of a NUMA node also limits the amount of DRAM available as cache to the Memory Mode. The assumption is that only the DRAM of the node will be used, especially since we know only the NVM of the node will be used if the process is restricted using `cpusets` [1]

[1]One of the tests contained a bug and was killed due to the system being out of memory. The OS killed the faulty test at about 255 GB of main memory usage, which is the amount of NVM on

The idea behind this test is not too complicated: first, we get a large area of memory. This area should be larger than the amount of DRAM available. We then access this whole area sequentially, and then we measure the time it takes to access a byte at the beginning of the area. If we can then see if this byte is still cached or not, we may be able to get some information about the cache size, or even the replacement strategy.

4.1.2 Cache Line Size Test

The goal of this test is to determine the granularity with which the Memory Mode swaps memory in and out.

For this test, we need a large chunk of memory, that is larger than the amount of DRAM available. Next, we clear the cache to create an area that we can execute the test in. We then measure two memory accesses that are *x* bytes apart. The idea behind this is that if *x* is larger than the granularity with which the Memory Mode swaps memory in, then both the first and second access will be slow. If, on the other hand, *x* is smaller than the granularity, the first access should be slow, while the second is fast.

We can already make a few assumptions about *x*: x is most likely a power of 2, and *x* is most likely not smaller than the L3 cache line size, which is 64 Bytes. We can restrict testing accordingly, by only testing for cache line sizes that are either 2^n or $2^n - 1$, and larger than 64 Bytes. This cuts down the time the tests take a lot, especially since we clear the cache once per *x* we test for.

4.1.3 Prefetch Test

The goal of this test is to gather information about any prefetching that the Memory Mode may do.

For this test, we need a large chunk of memory, that is larger than the amount of DRAM available. We clear the cache and create a testing area. We then access the memory with a certain stride sequentially. We chose 2^n and $2^n - 1$ as possible strides, up to 4096 Bytes, to cut down on run time and to only look at strides we believe are interesting. When we measure the time of these accesses, and compare it with the results from the cache line size test in Section 4.1.2, we may be able to see if an when prefetching happens, and if there are some boundaries over which no prefetching can happen[2]. The comparison with the results from the cache line

one NUMA node (and half of the system total).

[2]The CPU cache prefetcher for example cannot prefetch across page boundaries, because it only knows the physical memory, and two adjacent physical frames are not necessarily occupied by adjacent pages.

size test is necessary so that we do not mistake data that is always loaded together (i.e., in the same cache line) for prefetching.

4.1.4 Associativity Test

The goal of this test is to determine if the cache is n-way set associative, direct mapped, or fully associative.

For this test, we need an area of memory that is not in cache. To limit the amount of variables this benchmark has to test for, it uses the cache line size from Section 4.1.2 and the cache size from Section 4.1.1. If we know those variables, we can calculate, for a given set associativity, how large a set would be, and how far apart two memory lines belonging to the same set would be. Next, we access exactly the memory belonging to a certain set. We then access more memory than can fit into a set of the cache, but less than fits into all of the cache. Finally, we measure how long it takes to access the first few elements that we accessed.

The assumption here is as follows: if we guessed the right level of set associativity, the first entries will have been evicted from the cache, since the set was not large enough to store all the memory we accessed. If we guessed wrong, the memory we accessed would be cached in different sets, and since we accessed less than the total cache available, it should all still be in the cache. If we then look how fast those measured accesses are, we can determine if we guessed right or wrong.

Similar tests exist for CPU caches, they are for example shown in [18].

4.1.5 Writeback Test

The goal of this test is to analyze the write speed when using the Memory Mode to get information about the writeback behavior i.e., whether it is write back or write through.

For this test, we need once again a large chunk of memory, larger than the amount of DRAM available. We then write to the memory every x bytes, and measure the time it takes.

We are then looking if writes slow down at any point, or if they stay at a constant speed throughout the whole test. If they drop off at some point, that may be the because the Memory Mode needs to start writing back pages. If speed stays constant, it may be because pages are write through, and data is immediately written to the NVM and DRAM. It may also be that pages are written back asynchronously though, so further analysis may be required in that case.

4.2 Implementation Details

4.2.1 Cache Clearing

For multiple tests, we need a tool that makes sure that a certain memory region is currently *not* cached in DRAM.

The initial idea behind this tool was very simple: if you access enough memory, eventually something has to be evicted from the cache. In the first iteration, it was simply a for-loop that ran sequentially over a large chunk of memory, writing a `1` every 64 Bytes[3]. Imagine an example system with 64 GB of DRAM (and a much larger amount of NVM), on which you then run this for-loop over 128GB. Our assumption is that the start of the 128 GB region is now no longer in the DRAM cache, but instead resides in the NVM and has to be swapped in if accessed. If this assumption holds, we can now run a test in the region at the beginning of those 128 GB. Usually, this function is called between every test.

Since tests did not produce any useful results, we developed a more advanced method. Because we executed all tests in the same region at the start of those 128 GB, we assumed that the Memory Mode may recognize this region as some sort of a *hot region* that it tries to keep always cached, since there is a lot of activity there. The next approach was to select a random region within the large chunk of memory we use, that is large enough to execute the tests in it. This random region is then accessed first, followed by the memory before the test region in a descending manner, and the memory after the test region in an ascending manner. This pattern was designed to prevent the Memory Mode from recognizing a *hot zone* as above, as each test is executed in a random place, instead of the same place. It may also throw off the prefetcher a bit more, since the accesses are no longer strictly sequential and ascending, but rather ascending and descending interleaved.

4.2.2 Measure Memory Access Time

Measuring the time it takes to access memory exactly is very important for this kind of work, so this code will be described in a very detailed way. The code can be seen in Listing 4.2.3. We start by explaining the different building blocks used in the code:

- `__sync_synchronize()`: This function issues a full memory barrier. This prevents the compiler and the processor from moving memory instructions across it. [2]

[3]Since the CPU cache uses 64 Byte lines, it would not make much sense to write in shorter intervals.

- `_mm_clflush(void* ptr)`: This function invalidates and flushes the cache line that contains `ptr` from all levels of the cache hierarchy. [7]

- `__rdtsc()`: This function reads the processor's time stamp counter and returns it. [8]

Now, to explain the function itself: First, we flush the CPU cache lines that contain the `address` in line 8. This is to prevent us from measuring an access to the CPU cache instead of an access to DRAM or NVM, since we already know how the CPU cache works. We then read the CPU's time stamp counter in line 11. In line 14, we copy the value at location `address` into a local variable, that exists either in a register. Line 14 is also the main place where the code to measure a read access differs: in that function, a value from a register is written to the location `address` points to. Afterwards, we read the time stamp counter again. The code in line 20-21 is there to prevent the compiler from removing the memory access in line 14. This is done by returning the value retrieved from address in the `dummy` buffer. Line 14 now has a purpose and can no longer be omitted by the compiler. We then return the difference of the two time stamp counter values in line 23. The `__sync_synchronize` calls prevent any reordering of the instructions, as we need to ensure that the code is executed in exactly this order.

To ensure that everything is working correctly, we looked at the output of the compiler using `objdump`. This was necessary to ensure that the compiler did not remove or reorder any instruction that is crucial to the code.

4.2.3 Logging Tool

This tool is pretty straightforward: it takes the results of the benchmarks, usually (address, time) pairs, and writes them to a file.

We then visualize the data using `python` and `matplotlib`, to get a better understanding of the results we just produced. The simplest form of visualization is to plot the access times against the addresses using dots or crosses in a scatter plot. We also have some more complex features, like outlier elimination using the *z-score* [25], and drawing mean and variance instead of individual dots to better see linear correlation.

```
1  unsigned long measureMemoryAccess(
2          const char *address,
3          char *dummy)
4  {
5      unsigned long long cycles1, cycles2;
6
7      __sync_synchronize();
8      _mm_clflush(address);
9
10     __sync_synchronize();
11     cycles1 = __rdtsc();
12
13     __sync_synchronize();
14     char temp = *address;
15
16     __sync_synchronize();
17     cycles2 = __rdtsc();
18
19     __sync_synchronize();
20     temp += 1;
21     *dummy = temp;
22
23     return cycles2 - cycles1;
24 }
```

Listing 4.1: This is the code used to measure a read access.

4.2.4 Timer Interrupt Reader

One major source of interference for the time measuring would be an interrupt in the middle of the measuring section. Since timer interrupts are usually the most frequent interrupts[4], we decided to only check those. To track the number of interrupts, we open the `/proc/interrupts` file, which contains a table with the number of interrupts since system start for every interrupt and every processor. We retrieve the number of interrupts for the right core and interrupt type. We added this code to Listing 4.2.3 in line 9 and after line 19, to count the number of timer interrupts that happened during the time measuring section.

[4]If there is a lot of network activity or I/O on a system, those interrupts can be more frequent than the timer interrupt. Since we do not do anything other than running these tests on the system, timer interrupts are the most frequent on the system. This was also confirmed by looking at the `/proc/interrupts` file.

4.2.5 cgroups

To control the tests better, we used `cgroups` to restrict the test processes. `cgroups` are a Linux feature that can restrict what CPUs a given process can run on, and which NUMA node's memory it can use. Our tests were restricted to CPU number 8, which is the first CPU on the second NUMA node[5]. They were also only allowed to use the main memory of that node.

4.3 Results

In this section we will discuss the results of the benchmarks described in Section4.1.

To provide some reference for the times presented here, an access to DRAM takes about 300 cycles, while an access to NVM takes about 900. These numbers were measured using the measuring function from Section 4.2.

4.3.1 Test System

Motherboard	X11DPi-N(T) by Supermicro
DRAM	8 DIMM DDR4 16GB by Micron
NVM	4 DIMM Intel Optane DC Persistent Memory 128GB
CPU	2 Intel Xeon Silver 4215 CPU
OS	Fedora 31, 5.3.7-301.fc31.x86_64 kernel

Table 4.1: Detailed overview of the test system's setup.

The system has 512 GB of NVM and 128 GB of DRAM. It is split into two NUMA nodes, with each node having one CPU and half of the mentioned main memory. The system is running Fedora 31. The detailed setup can be seen in Table 4.1.

4.3.2 Cache Line Size Test

This test reads from two different addresses to see if they are both on the same cache line.

The results of this test are split into four graphs: the first two are histograms that show the distribution of the time it took for the first read. This read should always hit uncached memory. This is depicted in Figure 4.1. The figure contains

[5]There is no deeper reason as to why that CPU was chosen, except that it is on a different node than CPU 0, which may sometimes hold a special status.

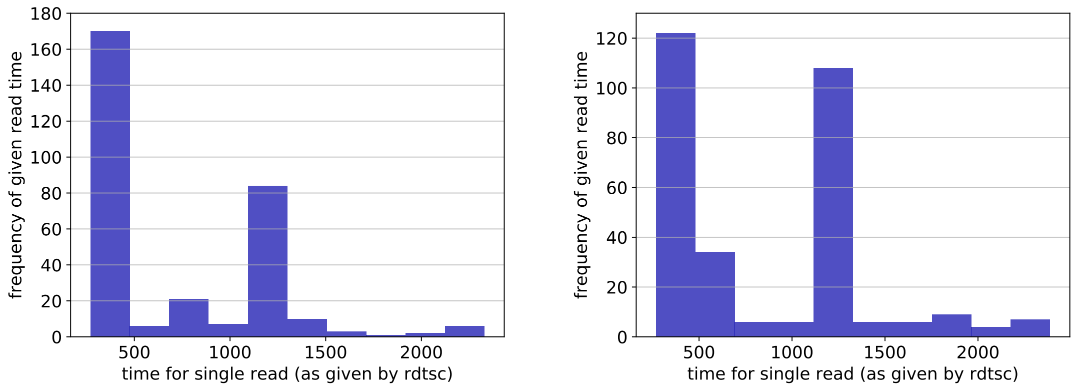


Figure 4.1: Distribution of the measurements of the first read for the cache line size test. On the left is the forwards test, and on the right the reverse.

a *forwards* and a *reverse* subfigure. This describes the order in which the two addresses were accessed. *forwards* means that `address` was accessed first, and `address + delta` was accessed second, while *reverse* is the opposite order. Figure 4.1 has two peaks, one around 300 cycles, and another around 1200 cycles. While it seems that the cache clearing did not work in most cases, it also seemed to have worked in a substantial amount of cases.

The second pair of subfigures in Figure 4.2 represents the average time it took the second read to come through. While there is some variance in the data, the averages are all pretty close to the 300 cycles line. These second accesses all (apart from very few outliers that were removed by the outlier detection) seem to be on cached memory.

The test starts with an assumed cache line size of 32 Bytes and goes up to 4 GB, so it starts with values that should absolutely be on the same cache line, and goes up to values larger than the largest page typically supported, which is 1 GB [6]. This leads us to the conclusion that something went wrong during the test, but we don't really know what.

4.3.3 Cache Size Test

This test accesses a lot of memory sequentially, and then measures how long it takes to read from a single address that is a certain distance from the end of the accessed area.

The results of this test are visualized in Figure 4.3. The test spans such a large area, because we were not sure if all of the system's DRAM (128 GB) was used as cache, or only half of it, since the test was restricted to the memory of one of

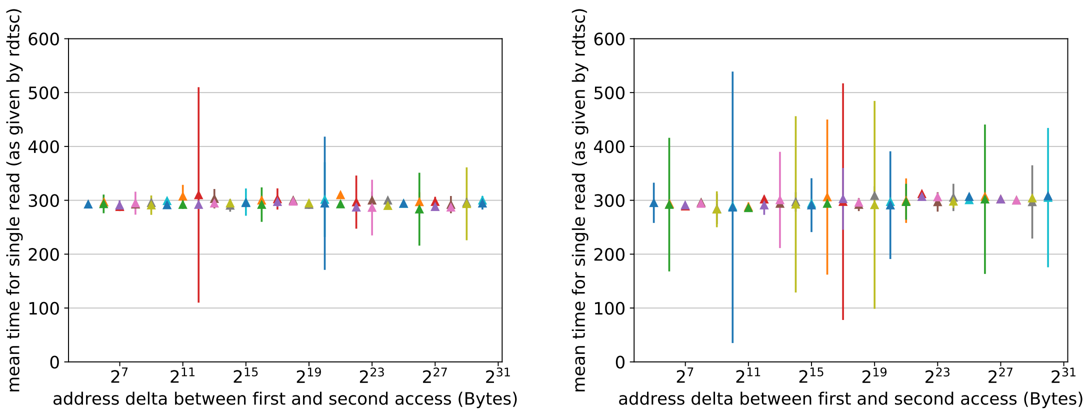


Figure 4.2: Mean time it took to read a byte from `first address + delta`, after `first address` was accessed. Left is forwards, right is reverse order. Triangle is the mean, while the length of the line symbolizes the variance. Triangles appear in pairs, as tests were executed for deltas of 2^n and $2^n - 1$.

the two NUMA nodes. As can be seen in the figure, it seems that almost all of the reads were on cached memory, apart from a few outliers. Ideally, accesses should have been on uncached memory if their distance to the end of the testing area is larger than the cache size. We can also assume that most of the accesses were to local DRAM, since accesses to remote DRAM are slower, and all accesses here take about the same time, which would indicate that only 64 GB of DRAM were used to defeat our test.

4.3.4 Prefetch Test

This test accesses memory that is initially uncached at regular intervals to see at what point the next access has already been prefetched.

This test relies on the assumption that the memory it is run on is not in cache, so that we can see a few initial slow memory accesses, which then speed up at some point (or not, depending on the level of prefetching). The problem is that the first access was in most cases already in cache when it happened. Figure 4.4 shows the distribution of the first memory access of each individual test run, and almost all values are concentrated around the 300 cycles mark, which indicates a DRAM access.

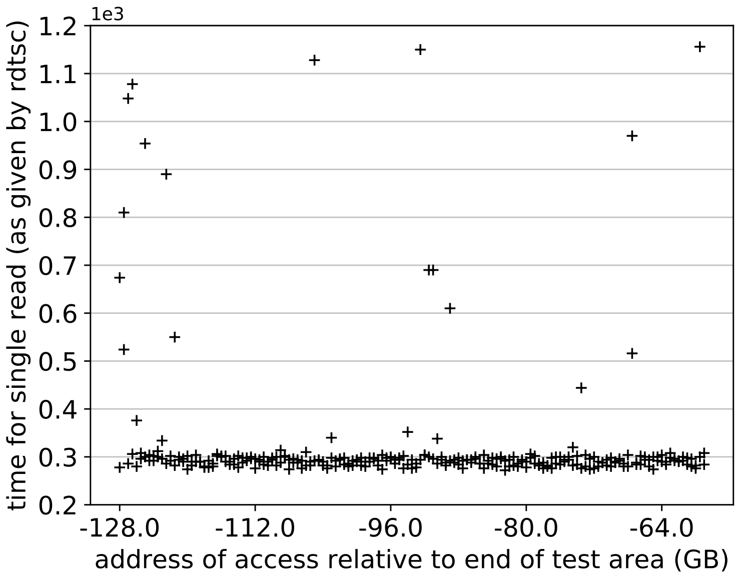


Figure 4.3: The results of the cache size test as a simple scatter plot. Note that the ticks on the *y-axis* are in scientific notation.

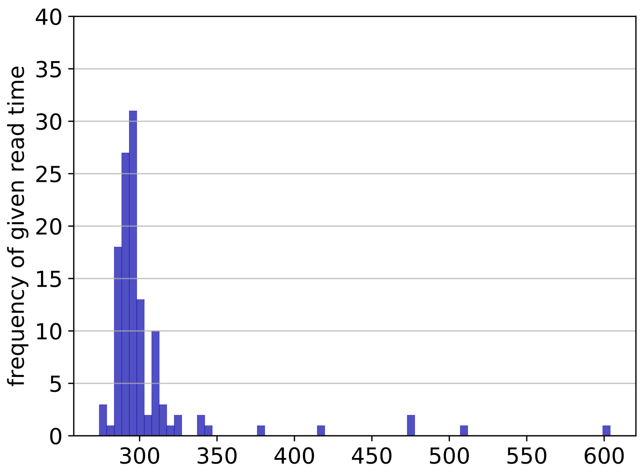


Figure 4.4: This histogram shows the distribution of the very first memory access of each prefetch test, for all strides we tested. Most values are concentrated around the 300 mark, which indicates an access to DRAM. The *y* axis indicates how often we ran the tests, and the specific values are not important.

4.3.5 Associativity Test

As mentioned previously in Section 4.1.4, this test takes the cache size and the cache line size as input. Since the results of the other tests that were supposed to determine those values were so inconclusive, this test was not run.

4.3.6 Writeback Test

This test measures writes at regular intervals over a large amount of memory.

The results of this test are visualized in Figure 4.5. This figures consists of two subfigures: the top one is a scatter plot containing the time measurements of periodic sequential writes over a large area. The writes were executed every 4 KB, but we "only" put every $16384th$ value into the scatter plot. Most of the crosses are on the 300 cycles line, which indicates write to DRAM. There seems to be no useful pattern in the few crosses above that line. The second graph is a histogram to show just how few crosses are not part of the big cluster at the bottom.

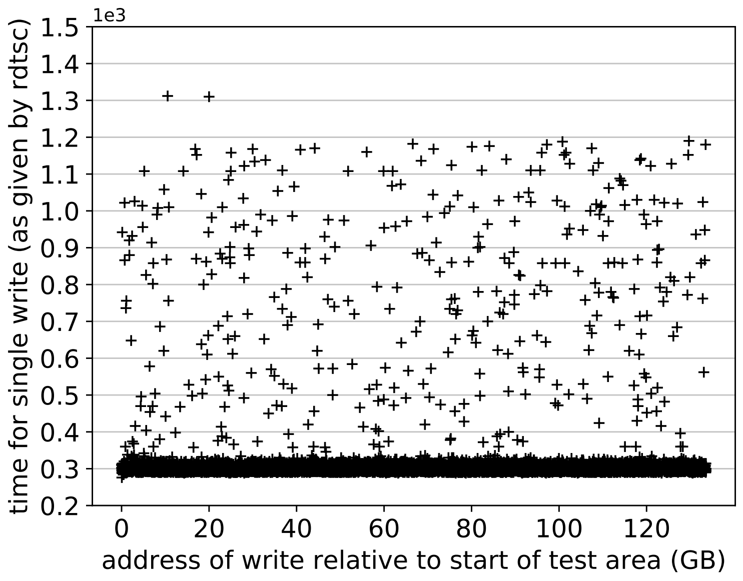
Unfortunately, we cannot really use these results to do any real predictions about the write back behavior.

4.4 Discussion

The results we found in this section are not very conclusive. We can nevertheless learn something from it. The Memory Mode seems to be more complex and efficient than initially assumed. The tests we described in Section 4.1 did provide expected results on my local test machine[6], when tested on the L3 cache. Now, it is clear that a small CPU cache that is designed to be small and extremely fast, is very much different than the Memory Mode, which handles hundreds of GBs of data. Nevertheless, it is still disappointing that the tests yield so few meaningful results.

We can speculate why the tests did not produce the wanted results. The cache clearing function may be a cause, although it was redesigned, so it is less likely to be the cause. Another reason may be the time measuring function: it could be that `_mm_clflush(address)`, which flushes any CPU cache lines containing `address`, also loads `address` into the DRAM. There could also be another problem with that function that we overlooked. The logging tool and visualization are less likely to be to blame, since we manually inspected the data using `hexdump`, and it looks reasonably varied, so it is unlikely that some data got erroneously duplicated, overwriting other data. There is also a lot of variance in some

[6]It has an Intel Core i5-4670K CPU, which has a 6144K L3 cache, that is 12 ways set-associative.



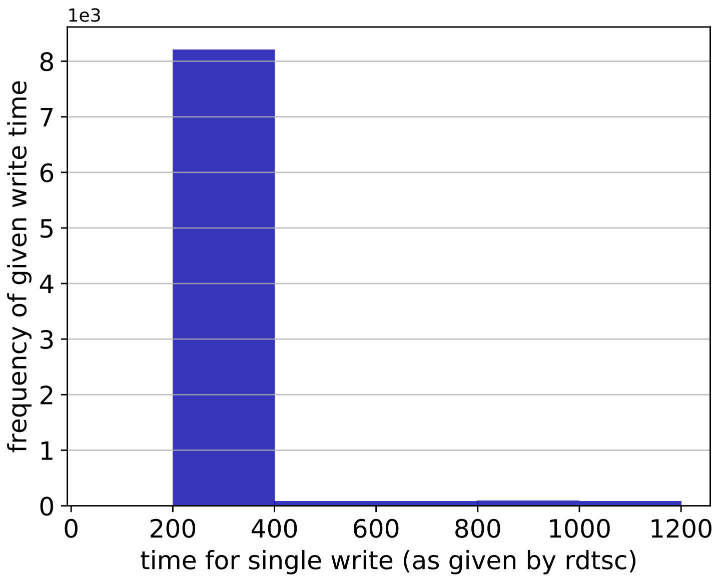


Figure 4.5: The first figure is a scatter plot of the time it took to write a byte to a certain address, relative to the start of the test. The black line at the bottom is not a line, but also crosses. The second figure shows the distribution of the values in the first plot, since the first plot may be a bit unclear.

results that we cannot explain, although the timer interrupts we recorded in about 50% of tests may explain some of it.

We can also speculate about the role of the CPU in the Memory Mode. Not every Intel CPU supports the NVM, and if you look at the marketing page for the CPU in our test system [11], there is a specific field to indicate if it supports Optane DC Persistent Memory. So my assumption is that the Memory Mode is not something that exists as an independent system, but rather something that is tightly integrated into the CPU. This might allow it to "beat" our tests, that assumed a more simplistic system.

Chapter 5

Reimplementation

In this part of the work, we will discuss the reimplementation of the Memory Mode in software. This is supposed to be a more adaptable and transparent alternative to the Memory Mode. In the first Section 5.1, we will discuss the design choices behind this implementation. In the second Section 5.2, we will discuss the actual implementation details and the problems we faced during the implementation. In the third Section 5.3, we will then discuss the performance of this reimplementation. Finally, we will discuss how the implementation went overall, and what could be done in the future to make it better in Section 5.4.

5.1 Design

This section discusses the general idea and design choices behind our reimplementation of the Memory Mode. Since the tests in the analysis Section 4 were inconclusive, we had to make some design choices ourselves that should otherwise have been determined by those tests. When making these choices, we either took an alternative that we thought made sense intuitively, or made the implementation easier. We will now go over the choices we made.

5.1.1 Overview

We start by providing a general overview of the project. The goal is to build a library that offers similar functionality to the Memory Mode. The library manages any memory the process allocates with it. The library has a limited amount of DRAM it can use as a cache, and a much larger amount of NVM it can use if the process needs more memory than there is DRAM available to the library. It has to decide which parts of the process's main memory are in the DRAM cache, and which parts are not. These decisions will be discussed in the next few sections.

The actual implementation, and the mechanics that are part of it, will be discussed later in Section 5.2.

5.1.2 Interface

This library provides its own implementation of `malloc`, `calloc`, and `realloc` functions from the C standard library [12]. These can be used to allocate memory that is then managed by this library. This means that we only manage a process's heap. While it would be nice to manage all segments of the process's main memory, the heap is usually by far the largest, so we think it is okay to only manage the heap in a first version of the library. The other segments (like the stack for example) are often not only smaller, but also very active, and it would therefore make sense to have them permanently in DRAM. Thus far, the library does not include these segments when it calculates how much DRAM it is already using as cache, since they are small, and calculating their size exactly is difficult.

5.1.3 Not Implementing `free`

As you may have noticed, the `free` function was missing from the list of functions the library provides. This was a deliberate choice. While `free` is of course important when designing regular programs, it is not *that* important in benchmarks. Most benchmarks allocate a lot of memory at the start, run the actual benchmarking code using that memory, and then only free it at the very end. `free` needs to be implemented in the future, if this library is ever used for anything other than benchmarks. But for now, we decided to reduce development time by not implementing `free`.

5.1.4 Cache Size

A major difference to the Memory Mode is that the amount of DRAM used for the cache is now configurable at compile and run time. This is not too difficult to do in our software implementation, and is one big advantage of the software variant. The Memory Mode always uses all the DRAM available in the system, as mentioned in the technical background Section 2.1.2.

Our library uses some local variables on the stack and global variables that are not accounted for in the amount of DRAM the library is allowed to use, but they only take up a limited amount of memory. We do not use recursion, so the stack only grows up to a certain upper bound, and the global variables are of fixed size, and do not grow either. The global variables include locks, file descriptors, and head and tail pointers for lists among others. Anything that can grow in size is allocated dynamically and taken into account.

5.1.5 Cache Line Size

An important design choice is to determine the granularity with which memory is moved between NVM and DRAM. There is a test in Section 4.1.2 that was designed to determine this, but was unable to do so.

So we chose 4 KB as the granularity. It is the smallest possible granularity we can work with, since `mmap` and `munmap` cannot handle a smaller granularity, and we need those functions to manage the DRAM [13]. Anything larger than 4 KB seems too expensive, since we also have to copy the entire amount of memory if only a single byte in the cache line is accessed.

5.1.6 Prefetching

Our library does not do any prefetching. This would be possible at page level, although we cannot see individual accesses to a page once it is in DRAM. Another problem is that the prefetcher would most likely run synchronously to the code, so it would only provide a benefit if copying two pages together from NVM to DRAM is much more efficient than copying a single page.

5.1.7 Write Behavior

The library uses write-allocate, meaning that a page is swapped into the cache when something is written to it. We can assume that this is similar to the Memory Mode, though we did not decisively prove it.

The library also uses a write-back policy instead of write-through. This means that changes to a page will only be transferred to NVM if the page is swapped out of DRAM.

5.1.8 Cache Inclusion Policy

There are two possibilities to store data when implementing something like the Memory Mode: an inclusive cache, where everything is always in NVM, and some parts are copied to DRAM, or an exclusive cache, where data is in either DRAM or NVM, but not both at the same time. When using the Memory Mode, the size of the main memory is equal to the size of the NVM, so we can assume that the Memory Mode uses an inclusive cache. Our library is an inclusive cache with a small change: since pages are only written back when they are swapped out as mentioned in Section 5.1.7, they only appear in NVM once they have been swapped out at least once. This means that freshly allocated pages only exist in DRAM, but a page that is swapped out and then back in will exist in both DRAM and NVM.

5.1.9 Associativity

The DRAM cache is fully associative, meaning that any part of the NVM can be put anywhere in DRAM. We do not control where we put data in the physical memory, we let the OS and the DAX file system on the NVM do that for us.

5.1.10 Replacement Algorithm

The library uses a *First In, First Out (FIFO)* replacement algorithm. This is a natural choice, since the library automatically knows when a page is swapped in, and does not gain any other information about the page while it is swapped in without some additional effort. The library could for example disallow access to certain pages that are swapped in to be notified when they are accessed, to implement some variant of a *least recently used* replacement algorithm, but that would require additional effort.

5.2 Implementation

This section goes over the implementation process of the library. It is split into four parts: the first part 5.2.1 covers the data structures needed to manage everything. The second part 5.2.2 covers the functions that do the actual work (i.e., move pages from DRAM to NVM and vice versa). The third part 5.2.3 covers the functions that will be called by a program using this library, such as `malloc`. The fourth and final part 5.2.4 describes some of the problems encountered during the implementation, and how we addressed them.

A clarification about usage of the phrases *swapped in* and *swapped out*. When using *swapped in*, we describe something that is currently in DRAM (and NVM). *To swap in* is then used to describe the act of moving something to DRAM, to achieve a *swapped in* state. *Swapped out* is the opposite, where something is only in NVM.

5.2.1 Data Structures

Here, we cover the data structures needed to manage the memory that is provided to a process by this library.

Swap File

We need a place to store the pages we move from the DRAM to the NVM. This is the swap file. It is a temporary file that resides in the DAX file system on the NVM. DAX file systems are explained in the technical Background Section 2.

Each page will have dedicated place in this file once it has been swapped out once. Pages that are swapped out for the first time are appended to the file.

This is also a place where `free` would cause additional work: when `free` is called for a certain memory region, all entries to the swap file from that region need to be removed. This means that we have make holes into the swap file. This then makes inserting new pages more difficult, since they can no longer be simply appended, but should instead fill holes, if there are any. While it is far from impossible to implement this, we believe it is not worth the effort for the prototype, as explained in Section 5.1.3.

The file is created the first time `malloc` is called.

Swap Lists

We need a place to store where in the swap file each page is. For this, we use two linked lists. One for all pages that are in DRAM, and one for the pages that are not in DRAM. The entries of these lists contain the starting address of the page in DRAM, and the offset in the swap file, if the page has an entry in the swap file. Linked lists may not the most efficient data structure for this purpose, but they are easy to implement, and elements can easily be inserted and removed. Each list also has its own read-write-lock, to control parallel access to them. This read-write-lock allows multiple threads to iterate over the lists to search elements in parallel, while also ensuring that only one thread can access the list while it is being altered.

Allocations List

This is list keeps track of all the memory regions that `malloc`, `calloc` and `realloc` have given to the process. This list was only added pretty late into development, when we noticed that we needed to implement `realloc`. Since `realloc` can extend or move existing allocations, it cannot be implemented without keeping track of the pointers returned by `malloc`. Before this list existed, not wanting to add it was another reason to forego the implementation of `free`. It stores three values per element: starting address, requested size, and allocated size. It also has its own mutex to control parallel access. We expect this list to be pretty short, so we chose a mutex over a read-write-lock.

5.2.2 Performing the Swap

This section explains the functions that move the pages between DRAM and NVM if necessary.

Swap In

This function is called to copy a page from NVM to DRAM.

It starts by retrieving the address of the page in NVM from the list of all swapped out pages. If it is no longer in the list, the function assumes that another thread has already taken care of this request and simply returns.

If the page has no entry in the swap file, this page is initialized with an anonymous page from `mmap`. These pages are completely zeroed [13]. This happens if the page is accessed for the first time after `malloc` is called.

If the page already has a corresponding entry in the swap file, the page should then be swapped in. This is done by allocating a new anonymous page with `mmap` at a random place, mapping the relevant part of the swap file with `mmap` to another random place, and then copying the data from the swap file to the newly allocated page in DRAM. Remember that the swap file resides in a DAX file system, so this is a copy from NVM to DRAM, even though both are mapped in the process's virtual address space. The new DRAM page is then moved to the right address with `mremap`. We do this to prevent other threads from accessing a page that is only partly initialized with the data from the swap file.

Finally, the list element containing the start of the DRAM page and the offset in the swap file is appended to the list of swapped in pages. This makes the replacement algorithm's job quite easy, since the swapped in list is automatically ordered from least to most recently swapped in. The replacement algorithm replaces the least recently swapped in page (also known as FIFO), as mentioned in Section 5.1.10, so it only has to take the head of this list.

Swap Out

This is the counterpart to the Swap In function from above. It removes a page from DRAM and stores its content in the swap file.

This function starts by retrieving the list element for this page from the list of swapped in pages. If the element is not in the list, we assume that another thread takes care of this and we simply return.

If the list element is retrieved, we check if the page has already a designated spot in the swap file. If not, we make the swap file longer with `ftruncate` and assign this additional space in the swap file to this page. This happens the first time a page is swapped out.

Now we can move the contents of the page to the swap file. This is done by duplicating the mapping to the page, so that it is now at a random place, and at its initial place. The mapping at the initial place is then replaced with a mapping to `/dev/zero` with no access rights. When this page is now accessed, a segmentation fault will be triggered, that is treated by the library's own handler. This handler is described in the next Section 5.2.3. We map `/dev/zero` to the page so that it is still reserved, but does not occupy any actual memory. We then `mmap` the designated part of the swap file, and copy the contents of the page. Both the swap file and the duplicate of the page in DRAM are then unmapped. We duplicate the mapping and remove the original to prevent other threads from accessing a page that is being swapped out.

Finally, the list element is appended to the list of all swapped out pages.

5.2.3 Functions called from Outside

This section will describe the functions that a process that uses this library will call. This includes the segmentation fault handler, so *calls* is used a bit more liberally.

Segmentation Fault Handler

Whenever the process accesses memory that is currently not in DRAM, a segmentation fault will be triggered. That signal is then handled by this handler. The handler is setup the first time `malloc` is called.

When the handler is invoked, it retrieves the address of the fault. The handler then checks if it needs to swap out a page before another page can be swapped in, and does so if necessary, using the swap out function. The page that is swapped out is chosen according to the replacement algorithm described in Section 5.1.10.

Then, the page that triggered the segmentation fault is swapped in using the swap in function. Two different threads may trigger a fault on the same page, and the handler is then called twice at the same time. This can lead to two pages being swapped out, but only one being swapped in. We believe that it is not a big problem, as the DRAM cache will be full again once the next page is swapped in.

malloc

This is the main function used to allocate memory when using this library. The first time it is called, it sets up the swap file and the segmentation fault handler as mentioned previously. `malloc` takes a single argument, which is the number of bytes the caller wants to allocate. Our implementation is a bit inefficient and only allocates multiples of 4 KiB, so each request is rounded up accordingly. We believe that this is acceptable for running benchmarks, since they usually allocate memory in large chunks before running the tests, so the internal fragmentation is limited, but it should be fixed in the long run.

`malloc` then calls `mmap` to map a region of the requested size at a random place. We map `/dev/zero` to the entire region, to reserve this part of the virtual address space. We also disallow any accesses to the region, so that the segmentation fault handler is called whenever a page is accessed for the first time. The handler can then initialize the page.

Finally, `malloc` creates entries in the swapped out list for each page, and also creates an entry in the list of pointers returned by `malloc`. Creating the entries in the swapped out list this early, instead of creating them lazily as memory gets accessed the first time, is necessary to eliminate race conditions between threads. If there was no list element before the page was created, the swap in function couldn't decide if the page was accessed for the first time, or if another thread took care of it.

calloc

This function is designed to allocate memory for arrays and takes two arguments: number of array elements, and size of array elements. Our implementation does a multiplication and then forwards the request to `malloc`. The contents of the region are then set to zero before the pointer is returned.

realloc

This function can be used to change the size of a memory region allocated with any of the three functions explained here. It takes two arguments: the address of the region, and the new size. We can look up the region in the list of all regions allocated with `malloc`. Three interesting cases are then possible:

- The new size is shorter than the old size: since we do not free any memory as explained in Section 5.1.3, we simply change the length of the region in the list and return.

- The new size is longer than the old size, but still smaller than the allocated size[1]: we can also simply change the size in the list and return.

- The new size is longer than the old size, and also larger than the allocated size: We would call free if it was implemented, and then we call `malloc` with the new size.

This means that, once more, the bulk of the work is done by `malloc`.

[1]We only allocate multiples of entire pages as mentioned in Section 5.2.3

Changing The Cache Size

As mentioned before in Section 5.1.4, with this library it is possible to change the amount of DRAM used at compile and run time. To change at compile time, set the `INITIAL_DRAM_CAPACITY` macro to the desired quantity. To change at run time, call the `mm_setDRAMAmount` function.

This function adjusts the amount of DRAM used by the library, and swaps out enough pages to meet the new goal if necessary.

5.2.4 Problems Encountered

Here, we will discuss some of the problems we encountered and how we addressed them.

Allocating Memory For Internal Data Structures

Two separate problems occurred when allocating memory for the internal data structures, such as the elements in the swapped in and out lists from Section 5.2.1.

The first problem appeared when we used our own `malloc` to allocate memory for one list element per DRAM page. Since our `malloc` only allocates entire pages, each list element now occupies an entire page. This is not only very wasteful, since list elements are only 40 Bytes in size, but it also leads to an infinite loop: the page for the list element also needs a list element to manage it, which then again needs a page from `malloc`, and so forth. We chose to solve this by using a slab cache. This slab cache takes a page, and splits it into many parts that can then each be used to store a list element. This prevents the loop where one page needs one entire page to manage it.

The second problem was that the slab cache initially called `malloc` to allocate a page for the cache. This leads again to an infinite loop, where some part of `malloc` uses the slab cache, which then calls `malloc` to get a page, and so forth. The solution we chose was to have the slab cache `mmap` its own pages. We also made these pages permanently resident in DRAM, for two reasons: the first being the use of linked lists, meaning we need the entire list very often, so any part that would be swapped out would most likely be swapped in again very shortly after. The second reason is a possible deadlock. Imagine page A containing the managing list element for page B, while page B contains the element for page A. If both were to be swapped out at the same time, they cannot be swapped back in, since both of the elements containing each others location in the swap file would be unavailable. While this second problem is not unsolvable, we believe it is far easier to keep the pages permanently in DRAM, especially considering the performance mentioned when explaining the first problem.

The slab cache pages are counted towards the amount of DRAM used by the cache. This could theoretically lead to a situation where the entire DRAM cache is filled with slab cache pages, although it is quite unlikely, since each slab cache page contains 102 elements, so the process would need to use about 100 times as much total memory as the size of the DRAM cache. Intel recommends a ratio of 1:8 for DRAM:NVM when using their Memory Mode, so our library should work fine with similar ratios [16].

External Libraries Calling malloc

Since our library should be able to handle multi-threaded processes, we used some `pthread` locks. This includes read-write-locks for the linked lists in Section 5.2.1 and some mutexes to prevent other small race conditions. Initially, these locks were initialized during the first call to `malloc` with the functions provided by the `pthread` library. Unfortunately, those functions call `malloc`, which leads to unwanted recursion, where `malloc` calls `pthread_mutex_init` which then again calls `malloc`. This means that the second call to `malloc` either runs the setup code again, which leads to an infinite loop, or it skips it and uses uninitialized locks. Both cases are quite bad. This was easily solvable though, since `pthread` provides static initializers for their locks, that initialize locks with default settings, without a function call.

Unfortunately, this was not the end of it, since some of the `pthread` functions to lock a lock also call `malloc` [3]. And of course, our `malloc` locks a lock at some point. This leads to an infinite loop, that only ends when the OS finally decides to kill the process because it uses too many resources. This was not as easily solved as the first problem, so we decided to implement our own locks that do not use `malloc`. While these locks do not use any advanced features that `pthread` locks may use, they work without `malloc`. An advanced implementation might try to make the library lock free to circumvent the entire problem, but that is still a lot of work.

Our `malloc` And System Calls

Another problem that arose when trying to run some benchmarks happened when one benchmark used `fscanf` to read a file. What appears to happen inside the function is that it calls `malloc` to allocate a page, and then immediately passes that to a `read` system call as a buffer. This is a use case that is probably not exclusive to `fscanf`, but that was the place where we observed it. This is a problem, since our `malloc` allocates inaccessible memory, that is only made accessible once a segmentation fault has been triggered, as explained in Section 5.2.3. `fscanf` does not appear to access the memory between calling `malloc` and `read`, so it is still inaccessible when the kernel receives the buffer. The kernel notices that the buffer is not accessible, and returns an error to the process, instead of writing to the memory, which would then trigger the wanted segmentation fault.

One unsuccessful idea was to overwrite the `read` function from `unistd.h` with a wrapper that accesses the buffer once before then calling the real `read`. `fscanf` didn't seem to call the overwritten version of `read`, so this did not work. This solution would also require more work to be complete, since we would need to overwrite the wrapper functions for all system calls that give the kernel a buffer to either read from or write to.

The workaround that got the benchmark running was to add a line at the very end of `malloc` that writes a zero ot the first page of the newly allocated memory, thereby triggering a segmentation fault, which puts the page into DRAM and makes it accessible. This is a workaround specifically designed for `fscanf` [2] and the benchmarks and *needs* to be addressed in the future. The workaround is not safe against race conditions, since it is possible that the page is swapped out again between `malloc` and the system call.

5.3 Performance

5.3.1 Performance of the Library

In this section, we will analyze the performance of the library in benchmarks, and then later analyze the reasons for this performance.

The Benchmark

We chose the Parsec Benchmark Suite [20] to measure the performance of the library, since it offers varied workloads. This benchmark suite is not enough to fully compare the library to the Memory Mode, as the Parsec packages only use workloads that are a few GB large [20]. The Memory Mode is in general quite difficult to benchmark, since it is not possible to limit the amount DRAM used as cache [3] as mentioned in Section 2.1.2. So you would need a benchmark that uses more memory than the system has DRAM available. It may be possible to limit the amount of DRAM using NUMA, although it would still be 64 GB in our case.

[2]Other system calls might need more than one page, but they did not pose a problem in our very specific case.

[3]You could open up the server and physically remove DIMMs from the motherboard to limit the amount of DRAM the system has, although that is not a flexible solution.

	Only DRAM	Our Library	Only NVM
blackscholes (700 MB)	0m 18,142s	0m 23,644s	0m 21,097s
freqmine (1.8 GB)	0m 28,439s	1m 24,830s	1m 6,800s

Table 5.1: The real time it took two different benchmarks to run on our system. Behind the name of the benchmark is the amount of memory it uses at its peak. Our library was given 1 GB of DRAM. *Only DRAM* and *Only NVM* mean that the benchmark was forced to use only that kind of memory as main memory.

Results of the Benchmark

The very limited amount of memory the Parsec benchmarks use are already enough to discover pretty serious performance problems in our library. The results can be seen in Table 5.1. Our library is outperformed by the *Only NVM* option, which means that our cache slows the system down instead of speeding it up. This makes the library useless in its current form. We will explore the reason for the problems in the next Section 5.3.2.

5.3.2 Reasons for the Performance

Since our library performed so poorly in the benchmarks, we decided to analyze where the problem lies.

If we look at the Flame Graph in Figure 5.1, we can see pretty clearly where the performance problem lies: the linked lists are the main time user. Functions starting with `_ll_` belong to our implementation of linked lists, and about 96% of the time is spent inside those functions. It might look like our read-write-lock might also be the problem, although that most likely also comes from the slowness of the linked lists: our implementation of the read-write-locks prevents new threads from acquiring the read part of the lock if one thread tries to get the write part. But the thread trying to acquire the write lock is slowed down, because it has to wait for all other threads to release their read-locks before it can definitely acquire the write-lock.

The `swapOut` function is not visible in the Flame Graph as it has no time intensive list operations: the replacement algorithm from Section 5.1.10 only removes the head of the swapped in list, and newly swapped in pages are appended to that list. Both operations are very fast, so there is no lock contention for that list. The swapped out list on the other hand is often the target of linear searches, which is really bad for performance.

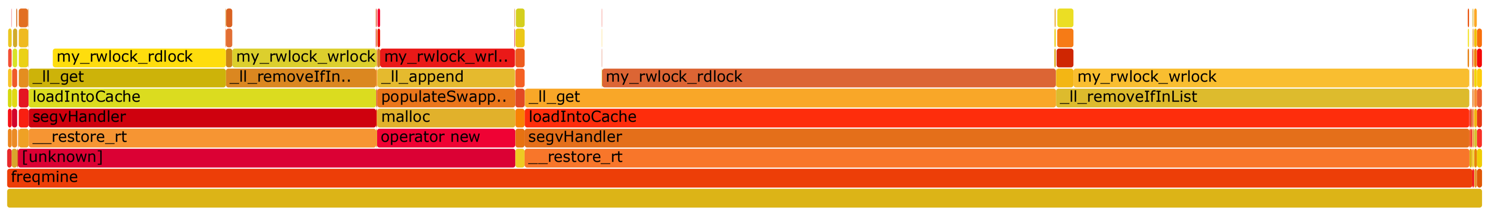


Figure 5.1: This is a Flame Graph [5] generated using `perf` [14]. It shows a call graph for the `freqmine` benchmark package of `parsec` with our library. Functions higher up were called by the function below them. The wider a box, the more time was spent executing that function. The top is cut off, but those boxes were very slim, so they are not as relevant.

5.4 Discussion

Generally, we believe the reimplementation worked out quite well. There are obvious shortcomings, but they are at least obvious, which makes it a lot easier to address them. We will now discuss the positives of the reimplementation, and also the negatives and possible future work.

While the results from Section 5.3.1 where quite disappointing, we still believe that the approach can work, especially since most of the performance problems came from a single source. This means that it could be possible to implement a Memory Mode in software, and even in user mode. Until now, stack and data segments are not yet covered by the library, but those segments are usually not as large as the heap, so adding them to the library should not be a large performance hit.

We have also learned a few lessons about potential problems when implementing such a project. We believe the main lesson learned is that it is quite difficult to use external libraries when implementing a function as crucial as `malloc`, since they may use `malloc` internally. This can then lead to loops, where our `malloc` calls a library function, which then again calls our `malloc`. Therefore, we needed to implement a few pieces software ourselves, instead of being able to take them from a library. We also learned that it is not too difficult to make the library thread safe. The race conditions and deadlocks that appeared could mostly be fixed without any major redesign operations.

There are two major things that did not work: the first one being the linked lists. Initially, they were only supposed to be a placeholder data structure that is easy to implement and not error prone, while we figure the other parts of the library out. Unfortunately, it took quite a bit longer than expected to figure the rest out, and then there was no time left to replace the linked lists. The one upside to this is that it is pretty easy to see what can be improved in the future. Replacing them with a hashmap, or even page tables should increase the performance massively.

The other problem are system calls: we have proposed a workaround in Section 5.2.4, which swaps in the first page of newly allocated memory when it is returned by `malloc`, to prevent other code from giving the kernel swapped out buffers, since the kernel doesn't trigger segmentation faults on them, but instead just rejects them. As mentioned when explaining the problem, the proposed workaround is still prone to race conditions, and is therefore not really satisfying. One possible solution would be to adjust all system calls that take a buffer, so that they trigger segfaults on buffers instead of rejecting them, if they are write protected. But that might pose other problems to the system, and it might be a lot of work to implement. So this is an open problem where the solution might not be as obvious as it is for the first problem.

A few smaller optimizations are also still possible: so far, `malloc` only allocates entire 4 KB pages. This is of course quite wasteful, and it would be good to update the code to be able to allocate smaller portions of a page. Another problem is that `mmap` can still be used by programs to allocate DRAM without our library noticing. Another future project could be to redirect `mmap` calls to our library as well. Then there is the implementation of `free`, which is still missing, but is necessary if we want to use the library for anything other than benchmarks. We also `mmap` and `unmap` the swap file each time we perform a swap operation. The virtual address space should be large enough to permanently map the swap file, to reduce the amount times we have to call `mmap` and `munmap`.

Chapter 6

General Conclusion

In this work, we analyzed Intel's Memory Mode for their NVM. They present it to us as a system that takes DRAM and uses it as a cache for the NVM, but do not tell us much more about the mechanics and policies behind it. We tried to find out how the Memory Mode works internally, and what policies are used. Then, we attempted to use this knowledge to build a software replica of the Memory Mode, to present a version of the Memory Mode that is more flexible and transparent.

Unfortunately, not everything went according to plan. In the analysis Section 4, the results of our benchmarks were very inconclusive and did not provide the insight into the Memory Mode we had hoped. We are still not certain what went wrong in that section, whether it was a single component of the implementation that did not work, or if the general design of the benchmarks was flawed, or if it was a combination of all those things. In the future, we may be able to develop other benchmarks with different implementations, an they may help us to understand how the Memory Mode works internally, and why our tests did not work.

This did not help the second part 5, where we tried to use the insights we gained in the first part 4 to reimplement the Memory Mode in software. Nevertheless, we build our own library to imitate the behavior of the Memory Mode. We encountered some problems during the implementation, but we were able to solve most of them. The performance of our library is still quite underwhelming, but here we do know what we can do to improve it: the main data structure needs to be replaced. There are other problems and projects for the future detailed in the discussion of Section 5.4. We believe that the library in its current state, together with the proposed future work is at least a partial success, as it shows that such a reimplementation of the Memory Mode should be possible, even in user mode.

Bibliography

[1] 128GB Samsung M386AAK40B40-CWD DDR4-2666 ECC DIMM CL22 Single - Mindfactory (Price converted from Euro to Dollar). https://www.mindfactory.de/product_info.php/128GB-Samsung-M386AAK40B40-CWD-DDR4-2666-ECC-DIMM-CL22-Single_1251689.html.

[2] 5.44 built-in functions for atomic memory access. https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html.

[3] Browse the source code of linux/tools/lib/lockdep/preload.c. https://code.woboq.org/linux/linux/tools/lib/lockdep/preload.c.html#alloc_lock.

[4] Direct access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[5] Flame graph github repository. https://github.com/brendangregg/FlameGraph.

[6] Hugetlb pages. https://www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html.

[7] Intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=_mm_clflush.

[8] Intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=rdtsc&expand=4545.

[9] Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[10] Intel Optane DC Persistent Memory Product Brief. `https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html`.

[11] Intel xeon silver 4215 processor. `https://ark.intel.com/content/www/us/en/ark/products/193389/intel-xeon-silver-4215-processor-11m-cache-2-50-ghz.html`.

[12] malloc(3) - linux manual page. `https://man7.org/linux/man-pages/man3/malloc.3.html`.

[13] mmap(2) - linux manual page. `https://man7.org/linux/man-pages/man2/mmap.2.html`.

[14] perf(1) - linux manual page. `https://man7.org/linux/man-pages/man1/perf.1.html`.

[15] Persistent memory overview. `https://docs.pmem.io/persistent-memory/getting-started-guide/introduction`.

[16] Quick Start Guide: Provision Intel Optane DC Persistent Memory . `https://software.intel.com/content/www/us/en/develop/articles/quick-start-guide-configure-intel-optane-dc-persistent-memory-html`.

[17] Persistent memory documentation: Namespaces. `https://docs.pmem.io/ndctl-user-guide/concepts/nvdimm-namespaces`, December 2019.

[18] Andreas Abel. Measurement-based inference of the cache hierarchy, 2012.

[19] Paul Alcorn. Intel Optane DIMM Pricing: $695 for 128GB, $2595 for 256GB, $7816 for 512GB (Update). `https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html`, April 2019.

[20] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[21] Integrated Circuit Engineering Corporation. Dram technology. `http://smithsonianchips.si.edu/ice/cd/MEMORY97/SEC07.PDF`.

[22] Mark Gottscho, Sriram Govindan, Bikash Sharma, Mohammed Shoaib, and Puneet Gupta. X-Mem: A Cross-Platform and Extensible Memory Characterization Tool for the Cloud. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 263–273, April 2016.

[23] Joseph Izraelevitz, Jian Yang, Lu Zhang, Amirsaman Memaripour, Yun Joon Soh, Subramanya R. Dulloor, Jishen Zhao, Juno Kim, Xiao Liu, Zixuan Wang, Yi Xu, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. August 2019.

[24] Allyn Malventano. How 3D XPoint Phase-Change Memory Works. `https://pcper.com/2017/06/how-3d-xpoint-phase-change-memory-works/`, June 2017.

[25] Mark Powers Siddharth Misra, Oghenekaro Osogba. Unsupervised outlier detection techniques for well logs and geophysical data.

[26] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-Efficient DRAM Caching Via Software/Hardware Cooperation. April 2017.