PERC: Persistent, Efficient, Recoverable, Consistent

by

William Anthony Mason

RESEARCH PROFICIENCY EVALUATION REPORT

Department of Computer Science

The University of British Columbia
(Vancouver)

October 2018

Abstract

Memory — the ability to save and recall information — is a fundamental characteristic of human endeavour and takes many forms. As we developed computing machines, we similarly developed mechanisms by which information could be stored for later use.

For modern computers, *drum memory* was the first manifestation of magneto-electric data storage and at the time of its introduction was used as both working memory as well as longer-term storage. Drum memory was replaced as working memory by *core memory*, which utilized magnetic wrapped cores for storing bits of information. Similarly core memory was in turn replaced by DRAM.

Each new class of memory exhibited faster performance but *different* behavior than the previous class. Drum and core memories were persistent, but core memory had a destructive write phase. DRAM memory was not persistent and required constant refresh to prevent the contents from decaying.

While faster, DRAM abandoned persistence of memory and gave rise to the separation of *memory* and *storage*. DRAM memories and processors became faster at a more rapid rate than storage became faster, further increasing the separation. While SRAM is faster than DRAM, it is much more expensive and only persistent as long as power is applied to it.

For decades, researchers have been searching for a new memory technology that is comparable in terms of performance and behavior to DRAM but *also* persistent. This achievement has proven to be elusive yet that has not discouraged the research community from considering how to exploit persistent byte-addressable non-volatile memory.

This report describes my findings while using the first commercial product to offer single level, byte-addressable non-volatile computer memory that behaves much like DRAM and observes how this behavior might impact development of systems that exploit this "new" class of memory.

Revision: October 18, 2018

Lay Summary

For the past 40 years, memory in computers has been primarily *volatile*, which means that the contents of the memory are lost when power is removed. In the past decade, memory that is *non-volatile* has emerged as high density, low cost alternative to traditional disk drives. Recent technology improvements have made it almost as fast as *volatile* memory. Its two key advantages are that it uses less power, which extends battery life in small devices and reduces power requirements in data centers, as well as providing ten times more memory in the same amount of space.

This report explores one such technolgy and seeks to find insights into how this new type of memory can be effectively exploited.

Contents

List of Tables

List of Figures

Glossary

CPU Central Processing Unit

DAX Direct Access eXtension

DDR4 Double Data Rate Fourth-Generation Synchronous Dynamic Random-Access Memory

DRAM Dynamic Random Access Memory

FRAM Ferroelectric Random Access Memory

MRAM Spin Transfer Technology Magnetic Random Access Memory

NRAM Non-volatile Random Access Memory base don Carbon Nanotubes

NUMA Non-Uniform Memory Architecture

NVDIMM Non-Volatile Dual Inline Memory Module

NVM Non-Volatile Memory

NVME Non-Volatile Memory Express

PCIE Peripheral Component Interconnect Express

PCM Phase Change Memory

PMDK Persistent Memory Development Kit

RERAM Resistive Random Access Memory

SRAM Static Random Access Memory

SSD Solid State Disk

Chapter 1

Introduction

Without memory, there is no culture. Without memory, there would be no civilization, no society, no future. — Elie Wiesel

A Brief History of Memory and Storage

Modern computer architecture — the *von Neumann* model — has evolved to systems that are fundamentally structured as shown in Figure 1.1, where the processor and memory are tightly linked to one another, while data storage is loosely coupled. This architecture reflects that *storage* has been much slower than *memory* since the introduction of core memory in the 1950s.

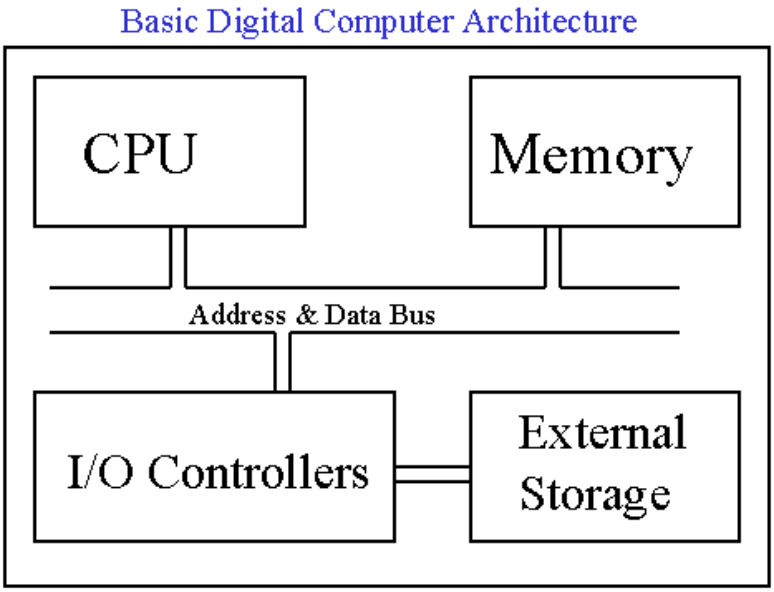
This was not always the case: *drum memory*, which predated core memory, was both main memory and storage. The introduction of core memory created this bifurcation, as core memory was faster but more expensive. Over time, core memory was itself replaced with DRAM [28]. Storage technologies moved from paper (punch cards and paper tape, for example) to magnetic media (tapes and disks).

Each of these divergent fields has undergone tremendous changes that did not remain in lock step with one another: performance and density have increased for both domains. In recent years the two domains have begin to converge once again, with storage moving to block-oriented non-volatile memories, such as flash memory. [5]

Memory technologies have continued to improve both in terms of performance — DRAM was itself eclipsed by SRAM [52] though though because SRAM remains considerably more expensive than DRAM, it is typically only used in performance critical areas of modern computer systems (e.g., the central processing unit itself).

Figure 1.1: Basic Computer Architecture

Source: Can Uger Ayfer, http://cayfer.bilkent.edu.tr/ cayfer/ctp203/review.html



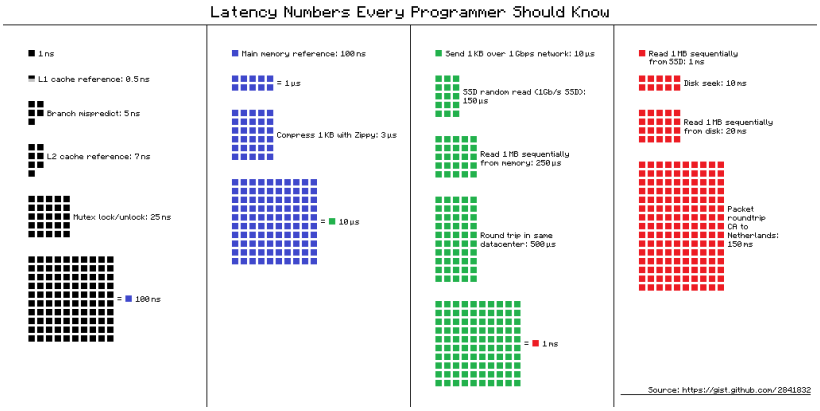
In 1965 Gordon Moore observed that the number of components that could economically be added to an integrated circuit was increasing rapidly, an observation that has come to be known as "Moore's Law". [69] While most often quoted in reference to processor technologies, the trend that Moore observed can also be observed in memory technologies, particularly as density has led to increased memory capacities.

While magnetic storage technologies benefited from some aspects of the improvements in integrated circuits, a fundamental characteristic of media was a significant delay in obtaining the data — the *latency* inherent in the physical movement of the equipment imposed a fundamental distinction that kept memory and storage separate in how they were handled by software using the system.

Around ten years ago, Jeff Dean, a well-known distributed systems expert at Google gave a presentation about the challenges of working with distributed systems. [22] One of the slides in that presentation provided a succinct summary of the amount of time to perform common operations within a computer system. Figure 1.2 provides a graphic representation of the relative difference between these operations. [7] For the purposes of this work, the important point is to note that the speed of accessing main memory at the time was approximately 100 nanoseconds.

Figure 1.2: Latency Numbers

Source: Jonas Bóner, https://gist.github.com/jboner/2841832



Reading 1MB of data from a disk required **20 milliseconds**, half of which was the latency of the physical disk drive hardware.

Since that time, storage has undergone a tremendous shift due to the rapid development and deployment of solid-state storage: *persistent memory* that has been used to construct devices which mimic the behavior of disk drives. In the span of roughly 10 years, the bandwidth of non-volatile memory devices has increased dramatically, which led to the introduction of higher-bandwidth interconnects between the storage device and the CPU. Currently, the highest speed interconnect NVME over PCIE has a maximum theoretical bandwidth of approximately 32GB/s.

Many of these changes have been driven by the increased density and decreased latency for persistent memories. The convergence of this process increasingly appears to be a model in which the boundary between *memory* and *storage* over-

Figure 1.3: NVME Performance

(a) Bandwidth **(b)** Latency

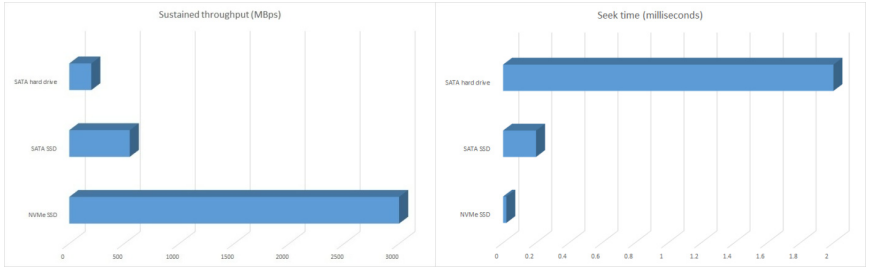
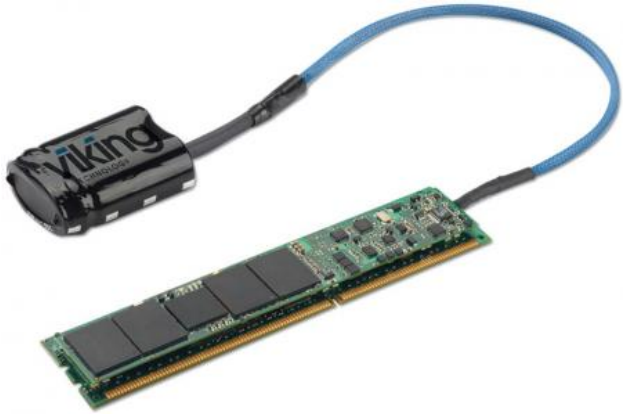


Figure 1.4: Battery-Backed Hybrid DRAM/Flash Memory

Source: RTC Magazine, http://archive.rtcmagazine.com/articles/view/102366



laps. While pragmatic engineering realities (ergo *cost*) make it likely that slower but cheaper storage options will continue to coexist with these new technologies, non-volatile memory technologies will be utilized in performance critical areas to improve overall system performance.

Thus, in recent years we have seen DRAM-like non-volatile memory solutions appear. Frequently, they have combined DRAM, non-volatile (flash) memory, and some backup power source — typically a large battery — into providing large capacity, high performance, byte-addressable, non-volatile memory, such as shown in Figure 1.4

This type of "storage class memory" is not widely used, given its disadvantages, which include cost, the need for external batteries, the lack of standardized support, and the relatively low density offered — often on par with DRAM.

At the end of May, 2018 — *after* the commencement of working on this project, in fact, Intel announced availability of their first *byte-addressable* non-volatile memory modules, with a form factor that makes them compatible with industry standard DDR4 memory. [21] While Intel has not confirmed the technology behind this new memory, there has been considerable independent analysis indicating that the underlying technology is PCM. [62]

The "hands-on" analysis provided in this report is based upon Intel "Apache Pass" NVDIMM form-factor memory. While interesting to have actual hardware to use in the evaluation, I have tried to focus on insights that are not tied to the specific product. This seems prudent because Intel and Micron recently announced they would be terminating their joint development program next year. [20]

Commercialization of non-volatile memory technologies for the production of

DIMM form factor persistent memory devices is an active area. These technologies include:

- PCM — This is the basis of the Intel/Micron developed 3D XPoint memory and forms the basis of the Intel Apache Pass product (which will use the Optane^{tm} trade name when released.)
- MRAM — STT-MRAM is being used in high capacity enterprise SSD devices from IBM, and is available in a DIMM form factor, but does not yet appear to have achieved high densities though it does show high performance. [39]
- FRAM — Ferroelectric Random Access Memory is commercially utilized non-volatile memory technology that has continues to be an area of active research in improving scalability. [67]
- RERAM — Resistive Random Access Memory, the technology based upon the *memristor*, continues to be yet another area of ongoing research.[49, 110]
- NRAM — Non-volatile Random Access Memory utilizes Carbon Nanotubes for data storage. These memories are presently in production and in use in specialized environments due to the novel thermal characteristics of Carbon nanotubes. [32]

Several of these are already used in specialized environment. For the most promising, the challenge remains commercializing the technology, often through scaling density or achieving economically viable cost/benefit levels.

In fact, it seems likely that several of these technologies will be used in the future for providing memory. While there is some value in evaluating the behavior of specific memory technology, it is somewhat ephemeral given the ever-changing nature of the technology industry.

It is more useful to find insights, even from such focused study, that provide a general sense of understanding about persistent memory. To that end, I have considered a number of aspects of the behavior of non-volatile memory:

- **Failure Models** — For any persistent data structure, it is imperative to understand the failure model of the domain. The challenge in this area is that storage experts are used to thinking of I/O related failure models, while processor behavior experts are used to considering consistency and correctness, but not persistence.
- **Performance** — storage systems are traditionally high-latency. There are write amplification issues that must be carefully evaluated. Memory systems

have concerns about cache behavior, the cost of consistency, and issues of memory locality because NUMA architectures create unequal costs for accessing specific memory.

- **Consistency** — persistence amplifies the cost of inconsistent state. A traditional way to recover from inconsistent machine state is to restart, which returns to a known-good state. When memory is persistent, inconsistent states do not automatically resolve when the system restarts.

I discuss failure models in §3.1, consistency considerations in §3.2 and insights gleaned from my work with NVM in §5.

Chapter 2

Related Work

The field of non-volatile memory is one with a long history. As noted in Chapter 1, early memories were, in fact, persistent and this led to models in which memory and storage were viewed as equivalent storage mechanisms. [19] The introduction of dynamic ram (DRAM) led to the bifurcation in storage technologies between high-speed but ephemeral ("memory") and low-speed but persistent ("media"). [28]

Our modern model of storage has evolved as well, from the punch cards of the late 19th century [92], to more modern media based magnetic devices, such as tape (ca. 1935) and other forms of magnetic media. [26, 35]

The introduction of flash memory [65] led to flash storage devices. Progress in this area has led to the emergence of solid state disk drives. [15] SSDs today are often the primary storage device of numerous computers, despite their higher cost than traditional hard disks, due to their performance characteristics.

The push to make SSDs faster has led to a rapid progression of bus technologies that permit exploitation of the increasing bandwidth and decreasing latency of such devices. The (re)-convergence of storage and memory has been demonstrated and discussed for more than 20 years.[41, 46, 68, 71, 73, 74, 100] The advent of PCM was one of those promising technologies. [10] The research community has explored this area extensively in the ensuing years in such topics as:

- Memory Management:[80, 95, 105]
- Data Structures:[13, 27, 45, 50, 51, 66, 75, 77, 88, 93, 96, 106, 111, 112, 113]
- Multi-threading: [36]
- Programming: [47, 48, 64]

- File Systems:[43, 57]
- Logging:[16, 90]
- Crash Consistency:[84, 99]
- Address Translation:[11, 55, 97]
- Database:[2, 3, 78]
- Checkpointing: [31, 107]
- Key-Value Stores:[14, 38, 44, 56, 103, 109]
- Transations:[63]

Prior work has been done with a variety of models for simulating the behavior of NVM. Indeed, my initial investigation was to determine the viability of investigating some of the considerations described in Chapter 1 via simulation. Ultimately, I was able to gain access to an Intel Apache Pass research system for performing this research.

Thus, one key difference between the prior work and the work described in this report is the use of actual hardware. As expected, that hardware does not behave as predicted. I will discuss this further in Chapters 4 & 5.

Chapter 3

Model

As noted in Chapter 1 having clear models for behavior in the system is an important part of searching for insight into how to effectively utilize NVM.

In this chapter, I will consider what it means for data structures to be *consistent* (3.2). Of course a key element of that understanding is to consider the potential set of *failure conditions* that might arise. This leads into the conversation about *failure models* (3.1). The challenge is that *consistency*, like magic, has a price. This is discussed further in §4.

In addition, I will also describe the methodology that I employed when evaluating the Intel Apache Pass NVM system.

Failure Models

There are 24 years of literature regarding non-volatile memory, some of which handles issues around appropriate failure models. Logically, this makes sense as persistent memory has the same type of failure requirements as storage based file systems: it is well-established that resiliency in the face of failure is an essential functional requirement for production systems. [76, 79]

Some of the papers which detail their solution to resiliency in the face of failure do not cite any specific failure model. [89, 97, 101] Further, a substantial number merely cite to generic failure classes such as "power failure" or "systems failure" — in other words a spontaneous reboot of the system at an arbitrary point of execution. [4, 6, 17, 34, 42, 54, 55, 59, 70, 82, 100, 108]

There are still a substantial number of papers that delve into the nature of failures. These papers cover a variety of issues, including:

transient persistent data in CPU cache — this scenario is one in which data is resident in a CPU cache at the time of failure. [8, 95]

byzantine failure — this is one in which components in the system either return corrupted data, or simply lose requests. [61]

partial failure — a situation in which some part of an operation succeeds while another fails. This manifests as torn or out of order write operations. [3, 9, 16, 18, 23, 36, 41, 74, 78, 78, 83, 104]

durability failures — what happens when the memory wears out, which can happen with some types of NVM. [24]

transactional memory — hardware and software transactional memory approaches to providing consistency models. [53, 87]

atomicity — the idea that a set of operations must occur all together or not at all [30, 40, 45, 63, 64, 75]

checkpointing — failure recovery from periodic safe spots (checkpoints) [85]

thread failures and delays — because of the pre-emptable nature of execution units (threads) on modern systems, there is a risk of having long gaps in time as the operating system switches between threads. Traditional techniques for synchronization such as locks do not work well in persistent memory systems. [33]

One challenge in constructing persistency techniques is the tools provided by the hardware has a significant impact on performance. [50, 93] Thus, a critical element in considering implementation of persistent data structures in non-volatile memory is balancing persistence requirements against the cost of forcing such persistence.

Analyzing failure in non-volatile memory is in fact a new class of failure. It is not just like disk storage failures, because of the processor cache interactions. Dynamic memory failures across reboot cycles are non-issues. Storage class persistence does not deal with *eviction* of data blocks under its control. These different semantics require a model of failure that considers both persistence and processor level issues. [77, 79]

Consistency

While §3.1 describes potential failure scenarios, in fact the primary concern is the ability to provide **consistency** guarantees. There is a dynamic tension between providing strong persistence guarantees, good performance, and generality of solutions. For example, a resilient non-volatile memory allocator does not provide

explicit guarantees about consistency of the data stored within the allocated memory without changing the usage model of that memory. One approach to providing stronger guarantees is to embed them within the programming language, either explicitly or via libraries. [46] However, this model requires changing the implementation of existing programs, which limits the likelihood of adoption.

Thus, consistency must define the level at which it operates and the guarantees that it provides relative to the failure model. Storage systems routinely are called upon to provide specific consistency guarantees that balance performance, generality, and persistence against one another. [12, 29, 91] This is not unique to storage, however, and the lessons for multi-processor consistency models is similarly applicable, with the added challenge of considering this behavior across system reboots. [1]

Methodology

Detailed results from my study of the Intel Apache Pass memory testbed system are reported in Chapter 4. I used three techniques to collect that data:

1. Intel Provided Tools (MLC) — see §3.3.1. This approach used the Intel Memory Latency Checker, along with the Intel provided AEP monitoring framework to analyze the baseline performance of the non-volatile memory.

2. Custom-Developed Micro-Benchmark — see §3.3.2. By using fine-grained (`rdtsc`) timing calculation, I observed the behavior and cost of the processor performing specific NVM-related memory operations.

3. Memory Allocation Measurements — see §3.3.3. By using an existing evaluation framework for non-volatile memory allocators, I measured the performance of existing memory allocators.

The system in question is described in greater detail in §4.1 and §4.2. The NVM was all accessed via DAX mode using either **xfs** or **ext4** with DAX mode enabled. Of the three configured NVDIMM modules, two were installed locally to node 0, and one was installed locally to node 1 (xfs was used with memory in both nodes, ext4 was used with memory in node 0).

The balance of this section describes details of the tools and the tests performed.

Intel Provided Tools (MLC)

The Intel Memory Latency Checker is a utility program that Intel makes generally available; I was provided access to the actual source code for the tool as part of the

Apache Pass access program. I used Version 3.5 of MLC, which is available on the Intel website. I did not modify the source code for this utility — my use was restricted to using it to understand what the checker was itself actually doing, as the test modes for persistent memory were not documented.

NVM testing was done using MLC via memory mapped access using a DAX supporting file system. The variables used were:

access type — memory access count be **random** or **sequential**. The specifics of the level of randomness are defined by the MLC utility. Buffer size for testing was 400MB.

test — tests included a read-only test, several read/write tests of varying ratios with both cached and non-cached (non-temporal) operations.

stride-size — the test allows controlling the data operations. While I tested stride sizes from 1 to 4096 bytes, I discarded the results below 32 bytes and above 2048 as the tests did not appear to be stable (e.g., they failed) outside those ranges. The reported figures are for that range.

processors — the test permits using a range of logical cores in testing. I limited my use to only a single hyper-thread per core. Processor 0 is used to measure load latency, while processors 1 to 23 were used to generate load. I varied the number of processors to observe the system behavior with varying degrees of memory contention. I collected some cross-node performance data, but do not report that information in this report.

For each run of the test, I used an Intel-provided monitoring tool that collects performance data from the invidual NVDIMM modules. Thus, for each test run I would start the monitor, run the test, then stop the monitor. That tool generated data files that were then fed into a custom-built version of gnuplot and detailed graphs of the NVDIMM behavior could be observed. This was useful in validating that the system was in fact only accessing a single NVDIMM, rather than a striped memory configuration, for example. I have not included those charts in this report because they do not appear to offer substantial insight.

The details of the specific switches used, and the specific workloads they represent is shown in Table 4.3.

Custom-Developed Micro-benchmark

My purpose in developing this micro-benchmark suite was to evaluate the performance of processors across a number of different tests:

baseline tests — each run measured the time to perform `write` over the test file, as well as the time required to perform `fsync`. Each operation was repeated 10 times.

linked list manipulation — a block of memory was initialized to consist of a series of forward references plus a monotonically increasing value (a `counter`). The forward references provide a mechanism for disabling effective prefetching, since the processor cannot prefetch until the address has been loaded from memory. There are up to seven subtests:

- list initialization
- list walk, with counter increment
- list walk, explicit prefetch operation for the next entry
- list walk, explicit prefetch operation, `clflush` after each write
- list walk, `clflush` after each write
- list walk, `sfence` after each write
- list walk, `sfence` after each full run of the list

Note that these references are all done from the same L1 hardware cache set (the pointers are offset by one page).

multi-cache line linked list — the block of memory is initialized and multiple cache sets are used and the CPU ticks are counted.

no-op test — measure the time required per operation to perform 1 billion `nop` operations.

cache flush tests — using the linked lists and counters again, this time with various patterns for flushing and fencing.

periodic cache flush tests — linked lists again, this time with various flushing, and periodic sfence behaviors, against the same cache set or across cache sets.

non-temporal move behavior — using non-temporal move instructions to perform data writes.

TSX tests — evaluating transaction abort rates and times required for various transaction lengths against the same associative cache set.

These tests were performed against a varity of memory block sizes. The linked list lengths were dictated by the number of memory pages in the provided buffer. This model made it simple to test both dynamic memory as well as non-volatile memory, simpy by creating a buffer via memory mapping using anonymous mappings (for DRAM testing) or specific file mappings (for NVM testing).

Each test was run multiple times (normally 100).

Results for this are reported in §4.3

Memory Allocation Measurements

The final type of testing that I performed was for memory allocation. I started with the memory allocator evaluation framework developed by the Hasso-Plattner Institute. See nvm_malloc for the original source code, and fsgeek_nvm_malloc. This in turn was based upon the work by [86] and the evaluation of their memory allocator.

The modifications that I made included allowing the framework to work on actual persistent memory and on the Fedora system installed on the testbed system. I also integrated two of the PMDK memory allocators into this framework.

This framework is a very simple evaluation of the allocator. In each of these tests the size of the allocation unit is randomly chosen over a range, using the C++ uniform distribution. For my testing I only used 64 bytes. The default is to perform each operation 100,000 times. These tests are:

alloc/free/alloc — in this test, one pass of allocations, then one set of free operations in FIFO order, then a second set of allocations.

alloc/free — in this test one pass of allocation and free operations are performed.

fast alloc — in this test one pass of allocation is is performed. Nothing is freed.

linkedlist — a series of entries are allocated and added to a doubly linked list structure.

recovery — this is a test of the cost to recover state from persistent memory; I do not report any results from this test in this report.

Note that this is a multi-threaded test; allocated objects are only used by the thread that allocated the memory. The allocation structures and memory pool are shared.

Results for this are reported in §4.7

Chapter 4

Results

Test Hardware

The primary system for testing was provided by Intel Research; this was physical hardware and I had console level access to the system — in fact, I reinstalled Fedora on the system at one point. My own test runs collected data on the system configuration. I capture that information here.

The Linux version (notably the kernel) was a current version at the time of testing:

```
# uname -a
  Linux intelsdp1044 4.17.12-200.fc28.x86_64 #1 \
  SMP Fri Aug 3 15:01:13 UTC 2018 x86_64 x86_64 \
  x86_64 GNU/Linux
```

It included support for NVM as part of the base release package — this was *not* a custom build.

The CPU information for the system is described in Tables 4.1 & 4.2.

Memory in the system consisted of 12 32GB DRAM modules and 12 249GB NVM modules. This information was displayed using the `dmidecode --type 17` to display information specific to the memory modules installed on the machine.

DRAM modules appeared like:

```
    Handle 0x0026, DMI type 17, 40 bytes
Memory Device
Array Handle: 0x0024
Error Information Handle: Not Provided
```

Table 4.1: Hardware Configuration Information (**lscpu**) Part 1

Characteristic	Value
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	96
On-line CPU(s) list	0-95
Thread(s) per core	2
Core(s) per socket	24
Socket(s)	2
NUMA node(s)	2
Vendor ID	GenuineIntel
CPU family	6
Model	85
Model name	Genuine Intel(R) CPU 0000%@
Stepping	5
CPU MHz	2899.999
CPU max MHz	3700.0000
CPU min MHz	1000.0000
BogoMIPS	4400.00
Virtualization	VT-x
L1d cache	32K
L1i cache	32K
L2 cache	1024K
L3 cache	33792K
NUMA node0 CPU(s)	0-23,48-71
NUMA node1 CPU(s)	24-47,72-95

```
Total Width: 72 bits
Data Width: 64 bits
Size: 32 GB
Form Factor: DIMM
Set: None
Locator: CPU1_DIMM_A1
Bank Locator: NODE 1
Type: DDR4
Type Detail: Synchronous
Speed: 2666 MT/s
```

Table 4.2: Hardware Configuration Information (**lscpu**) Part 2

Flags	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cdp_l3 invpcid_single pti intel_ppin mba ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb intel_pt avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts hwp hwp_act_window hwp_epp hwp_pkg_req pku ospke

```
Manufacturer: Micron
Serial Number: 18B132B8
Asset Tag:
Part Number: 36ASF4G72PZ-2G6H1
Rank: 2
Configured Clock Speed: 2666 MT/s
Minimum Voltage: 1.2 V
Maximum Voltage: 1.2 V
Configured Voltage: 1.2 V
```

NVM modules appeared like:

```
    Handle 0x0028, DMI type 17, 40 bytes
Memory Device
Array Handle: 0x0024
Error Information Handle: Not Provided
Total Width: 72 bits
Data Width: 64 bits
Size: 255680 MB
```

```
Form Factor: DIMM
Set: None
Locator: CPU1_DIMM_A2
Bank Locator: NODE 1
Type: DDR4
Type Detail: Synchronous Non-Volatile
Speed: 2666 MT/s
Manufacturer: Intel
Serial Number: 00000310
Asset Tag:
Part Number: 8089A2173800000310
Rank: 1
Configured Clock Speed: 2666 MT/s
Minimum Voltage: 1.2 V
Maximum Voltage: 1.2 V
Configured Voltage: 1.2 V
```

I have omitted all but the first instance of each for the sake of brevity; full logs are available.

Similarly, using the `lshw` command in Linux, I captured detailed information about the system. The following is the summary information about all memory installed in the system:

```
  *-memory
  description: System Memory
  physical id: 24
  slot: System board or motherboard
  size: 3380GiB
  capabilities: ecc
  configuration: errordetection=ecc
```

Similarly, this is the information about the first DIMM module:

```
*-bank:0
     description: DIMM DDR4 Synchronous 2666 MHz (0.4 ns)
     product: 36ASF4G72PZ-2G6H1
     vendor: Micron
     physical id: 0
     serial: 18B132B8
     slot: CPU1_DIMM_A1
```

```
        size: 32GiB
        width: 64 bits
        clock: 2666MHz (0.4ns)
```

Note that this provides different, but consistent information. For example, the serial numbers match between the two commands.

The NVM information:

```
  *-bank:1
       description: DIMM DDR4 Synchronous Non-volatile 2666 MHz (0.4 ns)
       product: 8089A2173800000310
       vendor: Intel
       physical id: 1
       serial: 00000310
       slot: CPU1_DIMM_A2
       size: 249GiB
       width: 64 bits
       clock: 2666MHz (0.4ns)
```

Again, further information has been omitted but is available.

Only three of the NVM modules were provisioned: two for node 0, one for node 1. These were DAX enabled, with xfs formatted for two of them, and ext4 for one of them.

```
# mount
/dev/pmem0 on /mnt/pmem0p1 type xfs (rw,relatime,seclabel,attr2,dax,inode6
/dev/pmem2 on /mnt/pmem2 type ext4 (rw,relatime,seclabel,dax)
/dev/pmem10 on /mnt/pmem10 type xfs (rw,relatime,seclabel,attr2,dax,inode6
```

*Note: I have omitted the non-*DAX *volumes here.* All three were mounted in DAX mode, ensuring that memory mapped files within those file systems were directly accessed by the test application.

The `ndctl` command was used to capture information about the NVM provisioning as well:

```
    ndctl list --namespaces  --human
    [
      {
        "dev":"namespace0.0",
        "mode":"fsdax",
        "map":"dev",
```

```
    "size":"245.11 GiB (263.18 GB)",
    "uuid":"f05281b4-dfa7-4f48-ab8b-41d1f54205aa",
    "raw_uuid":"c6c12d35-853d-418a-8602-846d3fd6091c",
    "sector_size":512,
    "blockdev":"pmem0",
    "numa_node":0
  },
  {
    "dev":"namespace2.0",
    "mode":"fsdax",
    "map":"dev",
    "size":"245.11 GiB (263.18 GB)",
    "uuid":"44071681-9ffa-4df4-bd51-7c51092a83e4",
    "raw_uuid":"ee045c93-52e0-4d5e-9670-7bc82ce691d6",
    "sector_size":512,
    "blockdev":"pmem2",
    "numa_node":0
  },
  {
    "dev":"namespace10.0",
    "mode":"fsdax",
    "map":"dev",
    "size":"245.11 GiB (263.18 GB)",
    "uuid":"a755e8a2-c6ad-426f-8841-cf393bc2b4f9",
    "raw_uuid":"2f6e6311-7a52-4fc3-ac97-763454d18370",
    "sector_size":512,
    "blockdev":"pmem10",
    "numa_node":1
  }
]
```


None of the NVM modules are striped (a potential configuration done via a software striping driver in Linux) and the sizes correspond to NVM in a single DIMM location.

Finally, I used the `ipmctl` utility, which is available as part of the various Linux releases (including Fedora 28, which I used). It is the *Intel persistent memory control* application.

```
# ipmctl show -topology
DimmID MemoryType Capacity PhysicalID DeviceLocator
```

```
0x0001 DCPMEM 249.6 GiB 0x0028 CPU1_DIMM_A2
0x0011 DCPMEM 249.6 GiB 0x002c CPU1_DIMM_B2
0x0021 DCPMEM 249.6 GiB 0x0030 CPU1_DIMM_C2
0x0101 DCPMEM 249.6 GiB 0x0036 CPU1_DIMM_D2
0x0111 DCPMEM 249.6 GiB 0x003a CPU1_DIMM_E2
0x0121 DCPMEM 249.6 GiB 0x003e CPU1_DIMM_F2
0x1011 DCPMEM 249.6 GiB 0x0048 CPU2_DIMM_B2
0x1021 DCPMEM 249.6 GiB 0x004c CPU2_DIMM_C2
0x1001 DCPMEM 249.6 GiB 0x0044 CPU2_DIMM_A2
0x1111 DCPMEM 249.6 GiB 0x0056 CPU2_DIMM_E2
0x1121 DCPMEM 249.6 GiB 0x005a CPU2_DIMM_F2
0x1101 DCPMEM 249.6 GiB 0x0052 CPU2_DIMM_D2
N/A DDR4 32.0 GiB 0x0026 CPU1_DIMM_A1
N/A DDR4 32.0 GiB 0x002a CPU1_DIMM_B1
N/A DDR4 32.0 GiB 0x002e CPU1_DIMM_C1
N/A DDR4 32.0 GiB 0x0034 CPU1_DIMM_D1
N/A DDR4 32.0 GiB 0x0038 CPU1_DIMM_E1
N/A DDR4 32.0 GiB 0x003c CPU1_DIMM_F1
N/A DDR4 32.0 GiB 0x0042 CPU2_DIMM_A1
N/A DDR4 32.0 GiB 0x0046 CPU2_DIMM_B1
N/A DDR4 32.0 GiB 0x004a CPU2_DIMM_C1
N/A DDR4 32.0 GiB 0x0050 CPU2_DIMM_D1
N/A DDR4 32.0 GiB 0x0054 CPU2_DIMM_E1
N/A DDR4 32.0 GiB 0x0058 CPU2_DIMM_F1
```

This captures the system configuration in a compact form, demonstrating the full amount of memory on the test system.

Intel Memory Latency Checker

The measurements in this section were collected using the Intel Memory Latency Checker, a tool developed by Intel to evaluate memory bandwidth and latency. Version 3.5 includes explicit support for evaluating DRAM and NVM. It is NUMA aware and can be used for evaluating both node-local as well as node-remote memories. [94]

I note that the provided documentation for this tool does not describe some modes of this tool that are, in fact, used in this evaluation. Further, actually enabling the correct testing mode for NVM is not easy to reproduce from the information available. Thus, when reporting specific results I have included the command line switches used as part of the testing.

Table 4.3: MLC switches used during testing

Switch	Effect	See Section
`-d`	Latency injection (seconds)	
`-t`	Test time (seconds)	
`-l`	Stride size (bytes)	
`-R`	Read-only workload	4.2.2, 4.2.2
`-W2`	Read-Write 2:1 Workload	4.9 4.8
`-W3`	Read-Write 3:1 Workload	4.10
`-W5`	Read-Write 1:1 Workload	4.11 4.12
`-W6`	Non-Temporal Write Workload	4.13 4.14
`-W7`	Read-Non-Temporal-Write 2:1 Workload	4.15 4.16
`-W8`	Read-Non-Temporal-Write 1:1 Workload	4.17
`-W10`	Read-Non-Temporal-Write 3:1 (Streaming Triad)	4.18

The tests considered in this section all used a zero injection latency (`-d0`,) as this represents the highest workload against the given memory — latency injection provides a simple mechanism for evaluating performance on workloads that are not pushing the bandwidth boundary. While I did perform evaluations with higher injection rates, I have chosen to omit that additional data from this report.

Each specific test run uses various flags to control the behavior. Unless otherwise noted, all tests used the `--loaded_latency` option, which means that the test utility is measuring the latency of the memory with some load imposed.

Note that for all tests, **core 0** is used for measurements. Thus, when we report 3 cores in use, two of them are generating load and one is performing the latency measurements.

For the various tests I use the switches as shown in Table 4.3.

Note that reported stride sizes were 16, 32, 64, 128, 256, 512, 1024, and 2048 in all cases. In a few cases I report 4096, but for many tests that stride size failed to work properly with the given test. Similarly, I tested sizes below 16 bytes but found that it frequently failed.

Sanity Check

Because the early results I was observing were surprising, I spent time to verify that I was using the tools properly by reproducing the original Intel results. This exercise was useful because it allowed me to identify undocumented options being used by Intel in their own evaluation, understanding how to enable their "persistent memory" mode in the tests, and validate that I was able to obtain comparable re-

Figure 4.1: Bandwidth Evaluation of Test System against Intel Reference

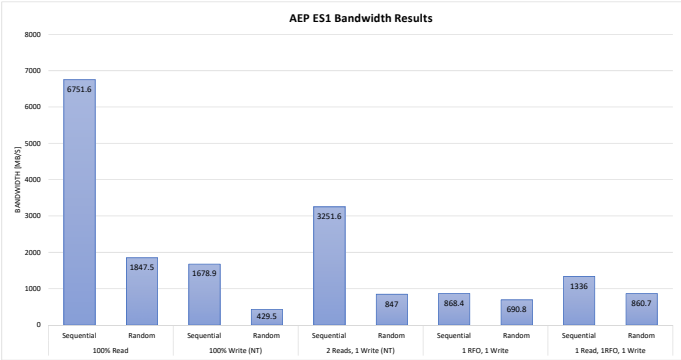


Figure 4.2: Idle Latency Evaluation of Test System against Intel Reference

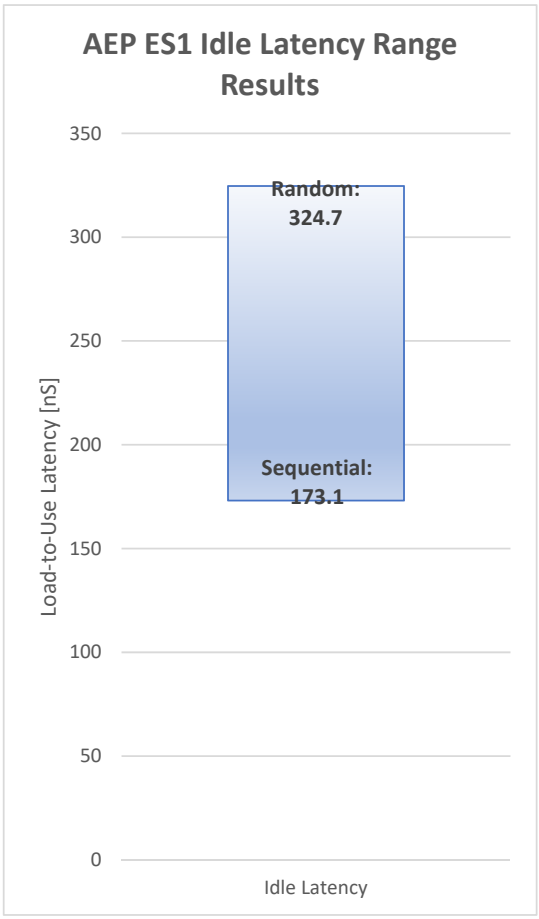


Figure 4.3: Random Read Load Latency Evaluation of Test System against Intel Reference

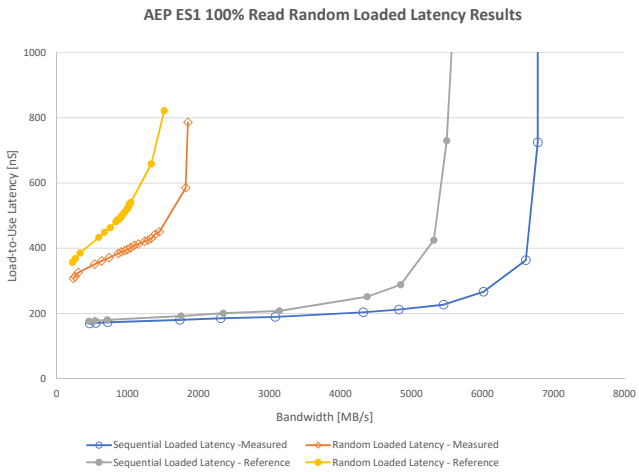


Table 4.4: Test System versus Intel Reference

Test Description	Figure
Idle Memory Latency	4.2
Loaded Memory Bandwidth	4.1
Random Read Latency	4.3

sults, suggesting that I was indeed testing the hardware appropriately. Ultimately, I did adjust my own data collection techniques to ensure that I was enabling persistent memory mode.

Intel provided benchmark numbers for three tests and the scripts to repeat their tests on the actual test system. These tests, and the results, are shown in Table 4.4. The results were slightly faster, as I was using a newer system, but within 20% of the original Intel reference numbers.

Figure 4.1 shows the bandwidth evaluation; the test system has somewhat better bandwidth than the Intel reference system.

Figure 4.2 shows the idle latency evaluation; the test system has somewhat better idle latency than the Intel reference system.

Figure 4.3 shows the random read evaluation for the test system. Again, it is somewhat better than the Intel reference system results.

These better measurements are indicative of the fact that the system under test was more recent hardware than the original Intel reference system. The improve-

ments are modest (approximately 10%) and seem to be consistent across the various tests. Thus, I concluded that my experimental setup was correct.

Evaluating how Intel was performing these tests provided me with insight into how Intel was using the **MLC!** test program. Notably they were using undocumented switches and generated specific configurations for testing that indicated specific optimization for their target benchmarks. For example, Intel allocates **two** cores ("threads") per NVM DIMM module for performing their load generation. As a result, in my own testing I used a varied number of threads to evaluate this area further.

Non-Temporal Baseline Measurements

This section describes the information for **non-temporal move** operations on the same node. Because these are done as non-temporal move operations, they bypass the cache and write directly to the actual memory. Note that I discuss the failure domain in greater detail in §3.1. The transfer involved here is sufficiently large that the impact of the memory controller caching does not impact behavior.

DRAM

Baseline testing was done using the switches:

```
--loaded_latency -d0 -t10 -W6 -l1024 -T
    -odata/bw_ctl_pmem0p1_seq_W6-0_23_400000_dram.dat
```

The file `data/bw_ctl_pmem0p1_seq_W6-0_23_400000_dram.dat` contained the confirmation information for the specific test layout:

```
0       W6 seq 400000 dram 0
1-23    W6 seq 400000 dram 0
```

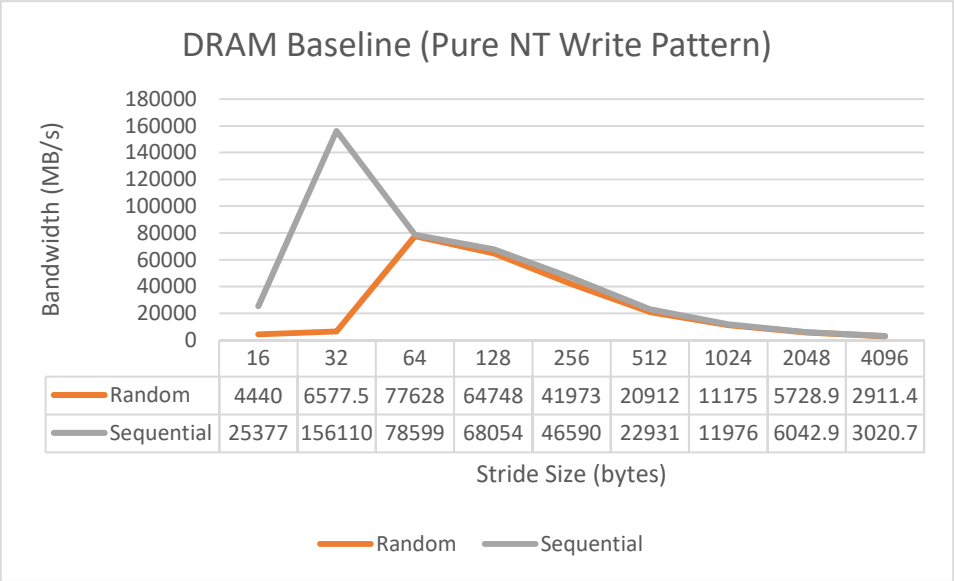
This drives the test to use core 0 for latency measurements, and cores 1-23 for load generation.

Note that the `-l` option was varied depending upon the "stride" size (unit of data handling). The control file specified the disposition of the individual CPUs

I do not report results for any other DRAM tests.

Figure 4.4 shows a baseline test for non-temporal read operations with various stride sizes. This establishes the "optimal" performance using dynamic memory. One core is used to measure the loaded bandwidth, the other 23 cores are used to generate load. It is interesting to note that a 32 byte stride size provides the best bandwidth measurement (1600000 MB/s sequential). I suspect this is due to

Figure 4.4: Baseline Measurement of DRAM Non-Temporal Write on the same NUMA Node



prefetching and caching behavior, though I did not validate this theory. The 64 byte stride size (one cache line) provides half this (80000 MB/s).

NVM

Baseline testing was done using the switches:

```
--loaded_latency -d0 -t10 -W6 -l64 \
  -odata/bw_ctl_pmem0p1_seq_W6-0_23_400000_pmem.dat}
```

The file `data/bw_ctl_pmem0p1_seq_W6-0_23_400000_dram.dat` contained the confirmation information for the specific test layout:

```
0    W6 seq 400000 pmem /mnt/pmem0p1
1-23 W6 seq 400000 pmem /mnt/pmem0p1
```

The `pmem` directive is used to put the test utility into "persistent memory" testing mode. The final value is the name of the directory to use. It **must** be a persistent memory to run this test. Otherwise the test will refuse to run.

Figure 4.5 shows a baseline test for non-temporal read operations. In this case the measurements show both NUMA local as well as cross-NUMA node persistent

memory values. Note that these values are substantialy below the DRAM values by more than an order of magnitude. The cross-node performance was surprising to me, as I would have expected the memory bandwidth to be the rate limiting issue, but apparently there is some consideration in the NUMA memory management that imposes a substantial performance restriction.

Equally surprising is that the cost of random versus sequential operations converge fairly quickly

Read

I tested specific configurations for random read and those are reported in this section in more detail than likely anyone will read. Note that these are cached reads.

Random

Random read testing was done using the switches:

```
--loaded_latency -d0 -t10 -R -l64 \
 -odata/bw_ctl_pmem0p1_rand_R-0_12_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_rand_R-0_23_400000_dram.dat` contained the confirmation information for the specific test layout:

```
0 R rand 400000 pmem /mnt/pmem0p1
1-12 R rand 400000 pmem /mnt/pmem0p1
```

The results from this test are shown in Figure 4.6. Note that the chart shows both bandwidth and latency in a single chart; bandwidth is along the left y-axis and latency is along the right x-axis.

Random read performance shows quite well at two cores, which suggests that may be why Intel performance figures have been computed using two cores for loading with 256 byte stride sizes.

These figures help better observe the nature of concurrency costs when accessing non-volatile memory.

Sequential

Sequential read is a common operation; it is also one that I would expect is an optimal case, benefitting from caches and prefetch logic.

Sequential read testing was done using the switches:

```
--loaded_latency -d0 -t10 -R -l64 \
-odata/bw_ctl_pmem0p1_seq_R-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_seq_R-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 R seq 400000 pmem /mnt/pmem0p1
1-20 R seq 400000 pmem /mnt/pmem0p1
```

Figure 4.7 shows the results from this test. The performance for sequential read is substantially better than random read (see §4.2.3), both in terms of bandwidth and latency.

Mixed Read/Write 2:1

This workload consists of a mixed read/write at two-to-one ratio and is considered for both random and sequential access patterns.

Random

Random read testing was done using the switches:

```
--loaded_latency -d0 -t10 -W2 -l64 \
-odata/bw_ctl_pmem0p1_rand_W2-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_rand_W2-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 W2 rand 400000 pmem /mnt/pmem0p1
1-20 W2 rand 400000 pmem /mnt/pmem0p1
```

Figure 4.8 shows the results from this test.

The results here suggest that bandwidth decreases with more cores, possibly due to some contention issues, but appears to be stable after 12 or so cores are active, though latency rises rapidly above four cores.

Sequential

Sequential read testing was done using the switches:

```
--loaded_latency -d0 -t10 -W2 -l64 \
-odata/bw_ctl_pmem0p1_seq_W2-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_seq_W2-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 W2 seq 400000 pmem /mnt/pmem0p1
1-20 W2 seq 400000 pmem /mnt/pmem0p1
```

Figure 4.9 shows the results from this test.

Bandwidth results are more consistent for this workload.

Mixed Sequential Read/Write 3:1

Sequential read/write testing using a three-to-one read-to-write ratio was done using the switches:

```
--loaded_latency -d0 -t10 -W3 -l64 \
-odata/bw_ctl_pmem0p1_seq_W3-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_seq_W3-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 W3 seq 400000 pmem /mnt/pmem0p1
1-20 W3 seq 400000 pmem /mnt/pmem0p1
```

Figure 4.10 shows the results from this test.

These results show a much larger lack of divergence than seen with prior workloads. Cache line impact seems to be fairly substantial with the 64 byte stride size showing markedly better performance than other stride sizes, while 128 and 256 byte stride sizes showing better latency than other stride values.

Mixed Read/Write 1:1

In this test, a one-to-one read-to-write workload is used with a random read/write access pattern; caching is eanbled. Results are provided for both random and sequential access patterns.

Random

The following switches were used:

```
--loaded_latency -d0 -t10 -W5 -l64 \
-odata/bw_ctl_pmem0p1_rand_W5-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_rand_W5-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 W5 rand 400000 pmem /mnt/pmem0p1
1-20 W5 rand 400000 pmem /mnt/pmem0p1
```

Figure 4.11 shows the results from this test.

Interestingly, performance here is remarkably uniform regardless of stride size.

Sequential

The testing was done using the switches:

```
--loaded_latency -d0 -t10 -W5 -l64 \
-odata/bw_ctl_pmem0p1_seq_W5-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_seq_W5-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 W5 seq 400000 pmem /mnt/pmem0p1
1-20 W5 seq 400000 pmem /mnt/pmem0p1
```

Figure 4.12 shows the results from this test.

Non-Temporal Write

The non-temporal tests explictly avoid the processor cache; as such they provide a better measure of pure performance for the underlying non-volatile memory. The tests in this case work for both random and sequential access patterns.

The workload used for these tests is a pure **write** workload. This is useful when considering predominately write usage patterns, such as for logs.

Random

In this test, a one-to-one read-to-write workload is used with a random read/write access pattern; caching is eanbled. The following switches were used:

```
--loaded_latency -d0 -t10 -W6 -l64 \
-odata/bw_ctl_pmem0p1_rand_W6-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_rand_W6-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 W6 rand 400000 pmem /mnt/pmem0p1
1-20 W6 rand 400000 pmem /mnt/pmem0p1
```

Figure 4.13 shows the results from this test.

Interestingly, performance here is remarkably uniform regardless of stride size. Performance generally seems best with 64 byte strides, with corresponding lower latency as well.

Sequential

Sequential read/write testing using a three-to-one read-to-write ratio was done using the switches:

```
--loaded_latency -d0 -t10 -W6 -l64 \
-odata/bw_ctl_pmem0p1_seq_W6-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_seq_W6-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 W6 seq 400000 pmem /mnt/pmem0p1
1-20 W6 seq 400000 pmem /mnt/pmem0p1
```

Figure 4.14 shows the results from this test.

The results here do quite well with 64 byte strides both in terms of latency and bandwidth. The performance drops with increased processor contention above 8 cores, with latency increasing substantially above 12 cores.

Non-Temporal Read/Write 2:1

This workload is in fact a **cached** read with **non-cached** workload mix. Writes are done using non-temporal instructions, while reads are done using CPU caching.

It uses a two-to-one read-to-write workload pattern. Non-temporal writes will perform cache invalidation, so a subsequent read of the memory region would force a reload from memory.

Random

The following switches were used:

```
--loaded_latency -d0 -t10 -W7 -l64 \
-odata/bw_ctl_pmem0p1_rand_W7-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_rand_W7-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 W7 rand 400000 pmem /mnt/pmem0p1
1-20 W7 rand 400000 pmem /mnt/pmem0p1
```

Figure 4.15 shows the results from this test.

The best bandwidth here is using 64 byte strides, with the lowest latency for any stride size. Bandwidth declines and latency increases with increased core contention.

Sequential

The testing was done using the switches:

```
--loaded_latency -d0 -t10 -W7 -l64 \
-odata/bw_ctl_pmem0p1_seq_W7-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_seq_W7-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 W7 seq 400000 pmem /mnt/pmem0p1
1-20 W7 seq 400000 pmem /mnt/pmem0p1
```

Figure 4.16 shows the results from this test.

The best bandwidth is seen using a 64 byte stride; latency is similar across the smaller stride sizes. Bandwidth decreases and latency increases with increased core contention for all stride sizes, although there is a peculiar dip in latency when half of the cores are active.

Non-Temporal Read/Write 1:1

This workload is a **cached** read with **non-cached** workload mix. Writes are done using non-temporal instructions, while reads are done using CPU caching.

It uses a one-to-one read-to-write workload pattern. Non-temporal writes will perform cache invalidation, so a subsequent read of the memory region would force a reload from memory.

Random

The following switches were used:

```
--loaded_latency -d0 -t10 -W8 -l64 \
-odata/bw_ctl_pmem0p1_rand_W8-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_rand_W8-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 W8 rand 400000 pmem /mnt/pmem0p1
1-20 W8 rand 400000 pmem /mnt/pmem0p1
```

Figure 4.17 shows the results from this test.

The 64 byte stride shows the best bandwidth and lowest latency. Bandwidth decreases and latency increases with increased core contention.

Sequential Streaming Triad Read/Non-Temporal Write 3:1

This workload is a streaming non-temporal write with triad read operations. There are three reads per write, with the reads being triads (sequential reads). This appears to be a common high performance load for certain types of video processing.

The testing was done using the switches:

```
--loaded_latency -d0 -t10 -W10 -l64 \
-odata/bw_ctl_pmem0p1_seq_W10-0_20_400000_pmem.dat
```

Note that the `-l` parameter was varied for different stride sizes, and the configuration file was varied to control the number of cores being used in the test.

The file `data/bw_ctl_pmem0p1_seq_W10-0_20_400000_pmem.dat` contained the confirmation information for the specific test layout:

```
0 W10 seq 400000 pmem /mnt/pmem0p1
1-20 W10 seq 400000 pmem /mnt/pmem0p1
```

Figure 4.18 shows the results from this test.

These results show best bandwidth for the 64 byte stride sizes. Latency is lowest with minimal processor contention. Other stride sizes show substantially lower bandwidth results.

Figure 4.5: Baseline Measurement of NVM Non-Temporal Write on the same NUMA Node

	16	32	64	128	256	512	1024	2048	4096
PMEM Local Random	409.5	350.3	428.8	413.3	396	396.8	395.4	392.6	390.3
PMEM Local Sequential	2033.6	1420.8	640.3	524	408.6	410.8	407.7	406.3	393
PMEM Remote Random	97	107	107.5	107.2	107.1	106.5	105.7	109.8	103.9
PMEM Remote Sequential	403.7	246.9	121.6	126.7	110.2	109.3	109.7	116.1	113.1

Figure 4.6: Random Read (R)

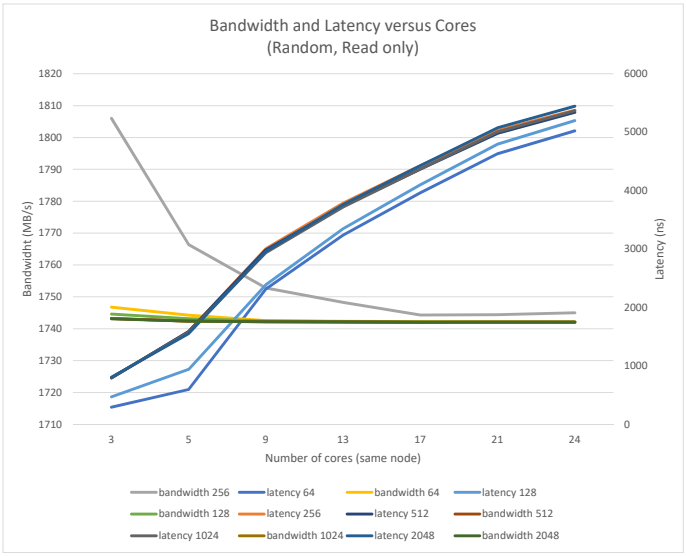


Figure 4.7: Sequential Read (R)

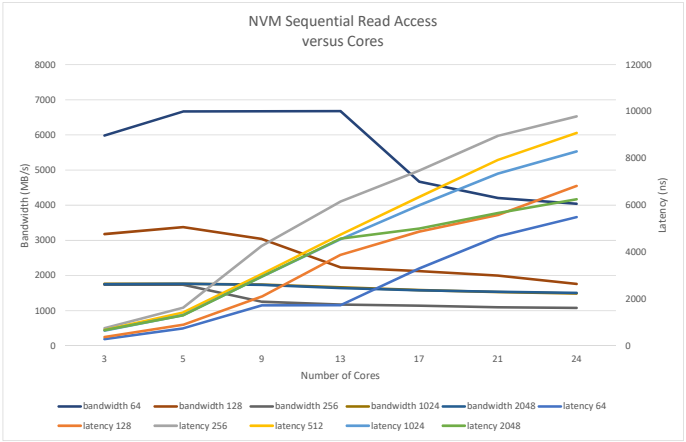


Figure 4.8: Random 2:1 Read/Write (W2)

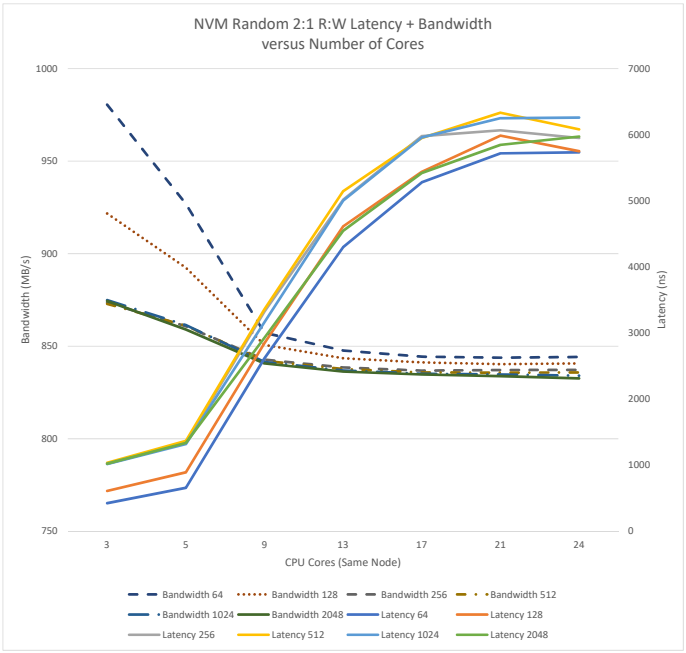


Figure 4.9: Sequential 2:1 Read/Write (W2)

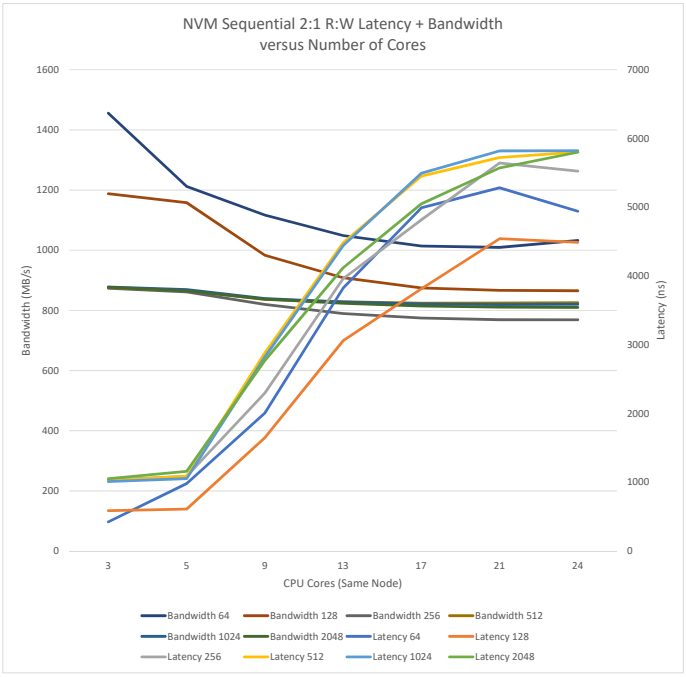


Figure 4.10: Sequential 3:1 Read/Write (W3)

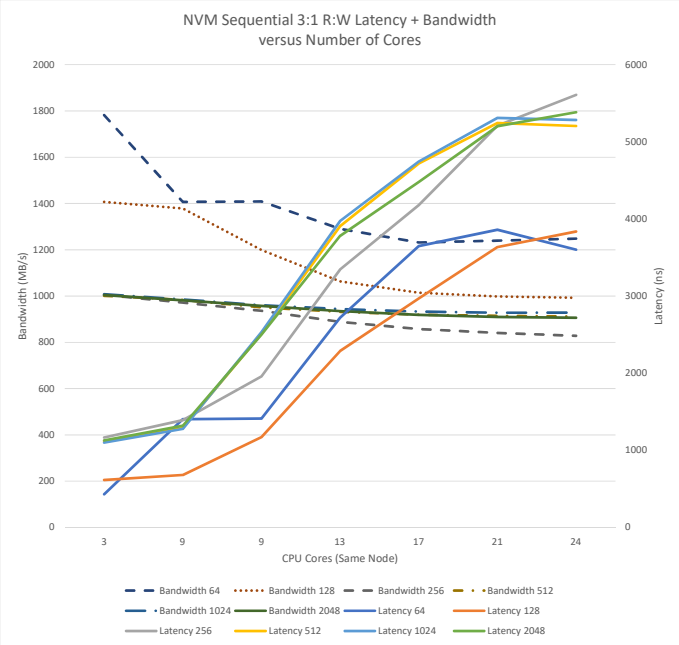


Figure 4.11: Random 1:1 Read/Write (W5)

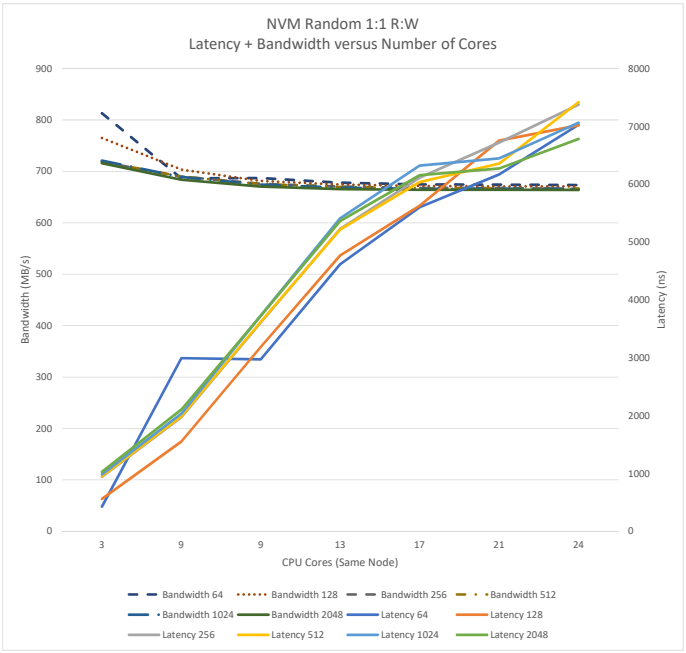


Figure 4.12: Sequential 1:1 Read to Write (W5)

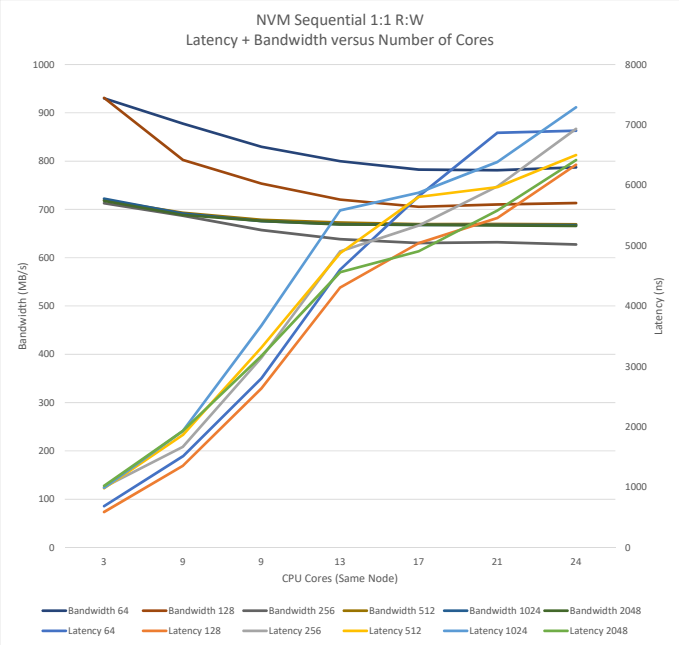


Figure 4.13: Random Non-Temporal Write (W6)

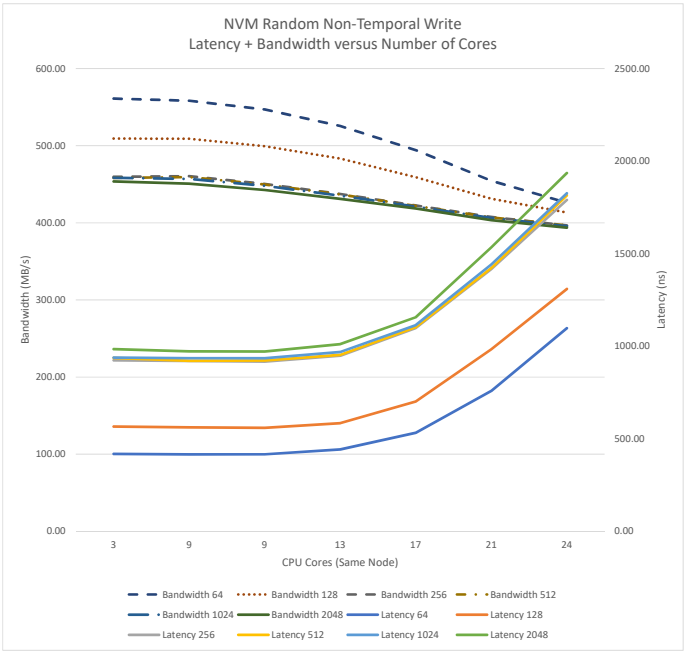


Figure 4.14: Sequential Non-Temporal Write (W6)

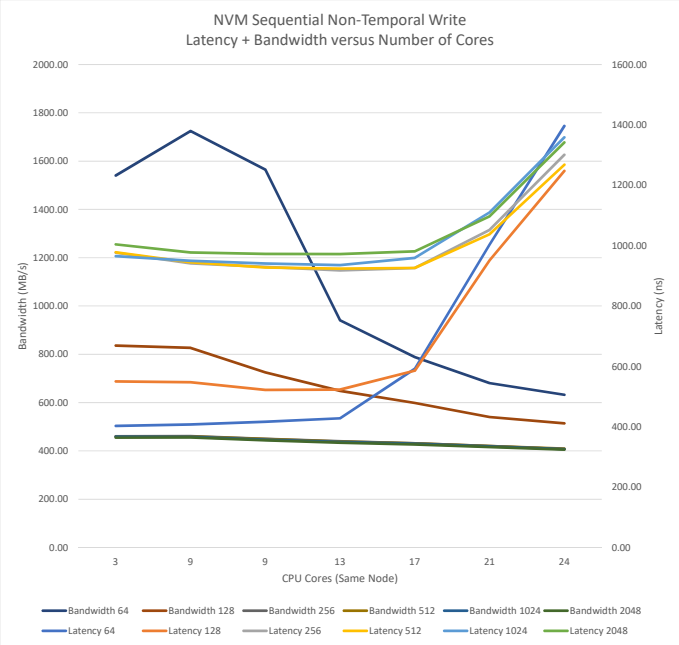


Figure 4.15: Random 2:1 Read to Non-Temporal Write (W7)

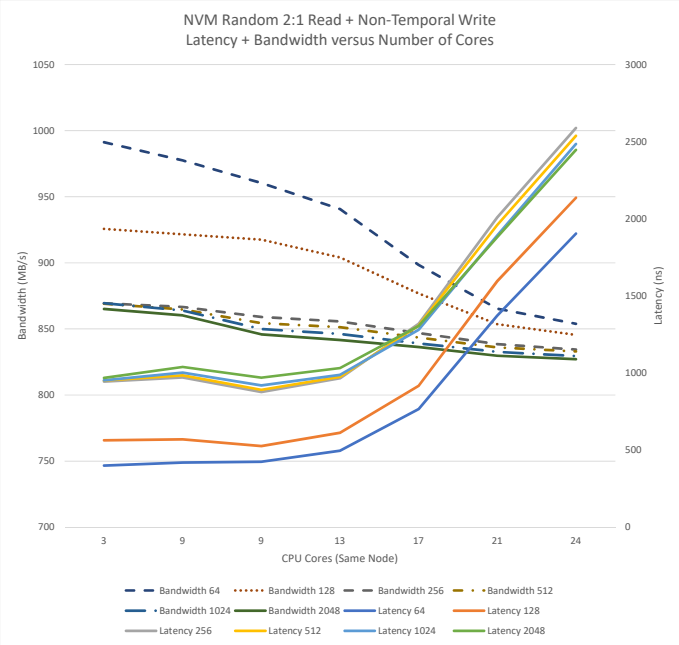


Figure 4.16: Sequential 2:1 Read to Non-Temporal Write (W7)

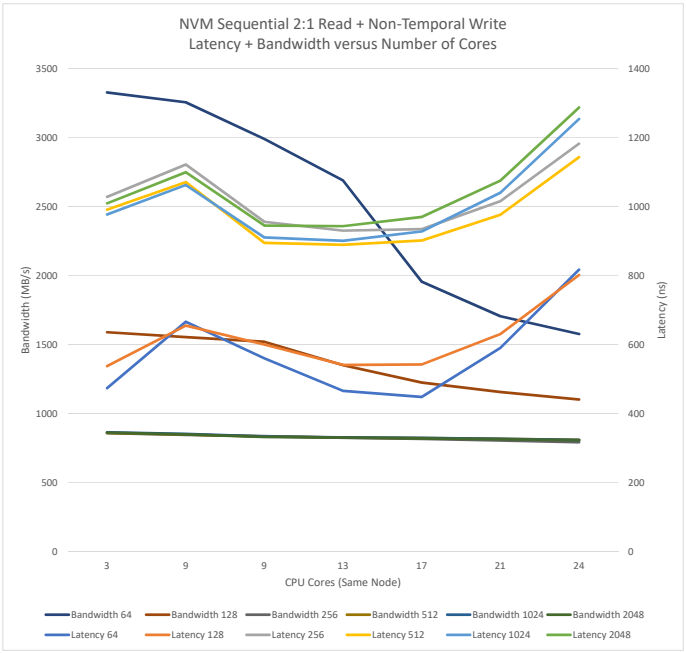


Figure 4.17: Random 1:1 Read to Non-Temporal Write (W8)

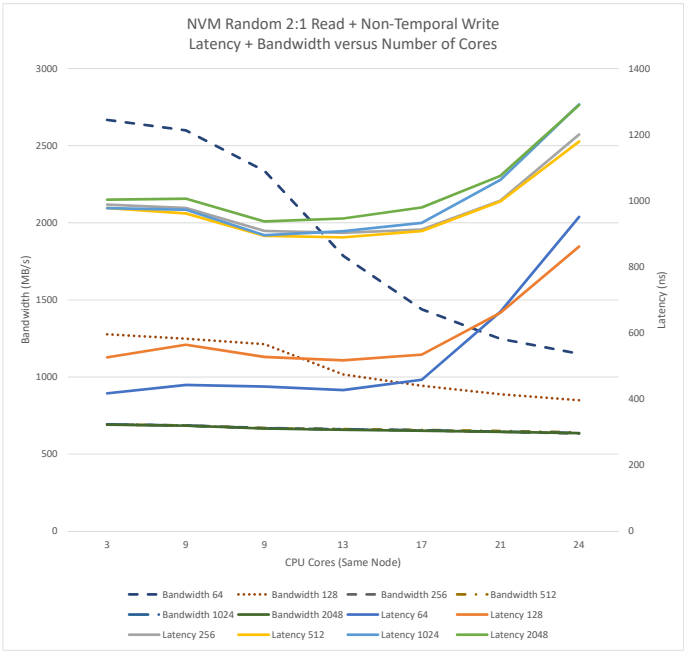
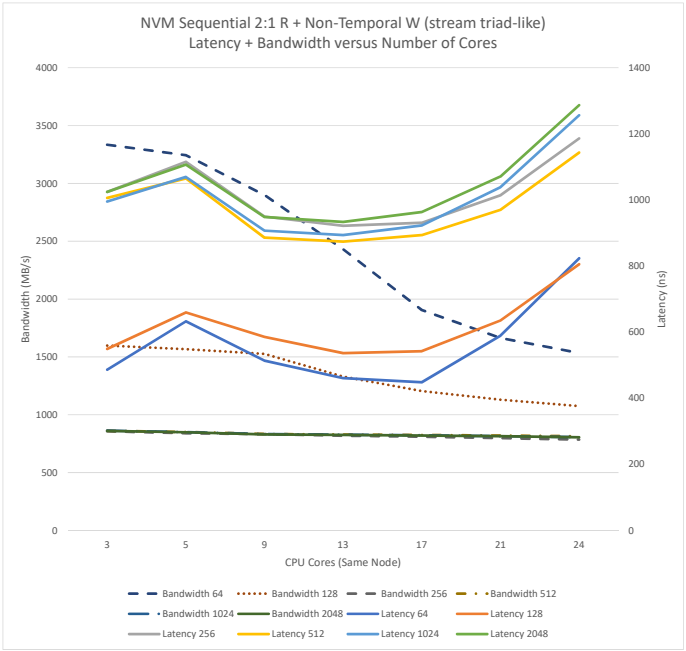


Figure 4.18: Sequential 3:1 Read to Non-Temporal Write (streaming triad) (W10)



Micro-Benchmark Results

The micro-benchmark results are based upon custom tests that I wrote as part of this investigation. My focus in looking at this performance was to better understand the interplay between cache behavior and persistence operations. By better understanding the trade-offs involved, I can more easily reason about constructing persistent data structures.

Throughout this section, I describe various flushing operation. These are achieve by using specific instructions, including the cache flush instructions: `clflush, clflushopt, clwb` and the memory barrier instruction `sfence`, which provides guarantees that all prior store instructions are now visible to other processors. **Note:** the `sfence` instruction does not guarantee the data is in the memory. At the present time, this typically means that it is visible to the *memory controller*. Current Intel platforms guarantee that once presented to the memory controller it will be persisted to main memory **even in the face of a power failure condition.**

Cache Sets

Many of the tests distinguish between the cache set that is being evaluted. The code does this by constructing linked lists of entries. As the linked list is traversed, a counter value is incremented — this is the *write* operation that causes a dirtying of the cache line.

On most Intel CPUs (and certainly the ones I have tested) the entries are scattered across cache sets. However, Intel's architecture guarantees that memory at the same offset ends up in the same associative cache set; my testing included data not reported here that confirms this is the case.

Many of the tests distinguish between testing linked lists allocated from the same or different cache sets. I describe how this was implemented in the remainder of this subsection.

Same Cache Set

Placing the list entries into the **same** cache set ensures that an n^{th} write will cause one of the other cached entries to be pushed back to memory (where n is the size of the associative cache set); I do not know *which* entry is written back. Note that when combined with fences, it does ensure that a consistent **set** has been written back, since doing otherwise would violate the fence guarantees.

```
static void
init_cache_test_memory_same_set(
    const unsigned pagecount,
```

```
    void *memory)
{
    record_page_t *recpages = (record_page_t *)memory;

    memset(memory, 0, pagecount * PAGE_SIZE);
    for (unsigned index = 0; index < RECORDS_PER_PAGE; index++) {
        record_t *r = &recpages[0].records[index];

        for (unsigned index2 = 0; index2 < pagecount; index2++) {
            uintptr_t diff1, diff2;

            r->s.next = &recpages[(index2 + 1) %
                        pagecount].records[index];
            r->s.counter = 0;
            diff1 = (uintptr_t)r - (uintptr_t)r->s.next;
            diff2 = (uintptr_t)r->s.next - (uintptr_t)r;
            r = r->s.next;
        }
    }

    for (unsigned index = 0; index < RECORDS_PER_PAGE; index++) {
        record_t *r = &recpages[0].records[index];

        do {
            r = r->s.next;
            assert(NULL != r);
        } while (r != &recpages[0].records[index]);
    }
}
```

This generic code sets up the "same cache set" and is used to test in the various configurations.

Different Cache Set

For these tests, cache memory is initialized to form a pattern that will not place a list entry in the same processor cache set. It creats a diagonal pattern through memory. The code for this is:

```
static void
```

```
init_cache_test_memory_different_set(
    const unsigned pagecount,
    void *memory)
{
    record_page_t *recpages = (record_page_t *)memory;

    memset(memory, 0, pagecount * PAGE_SIZE);
    for (unsigned index1 = 0; index1 < RECORDS_PER_PAGE; index1++) {
        record_t *r = &recpages[0].records[index1];
        for (unsigned index2 = 0; index2 < pagecount; index2++) {
            unsigned offset = (index2 + 1) % pagecount;
            r->s.next = &recpages[offset].records[(index1 +
                        offset) % RECORDS_PER_PAGE];
            r->s.next);
            r = r->s.next;
        }
    }

    for (unsigned index = 0; index < RECORDS_PER_PAGE; index++) {
        record_t *r = &recpages[0].records[index];

        do {
            r = r->s.next;
            assert(NULL != r);
        } while (r != &recpages[0].records[index]);
    }
}
```

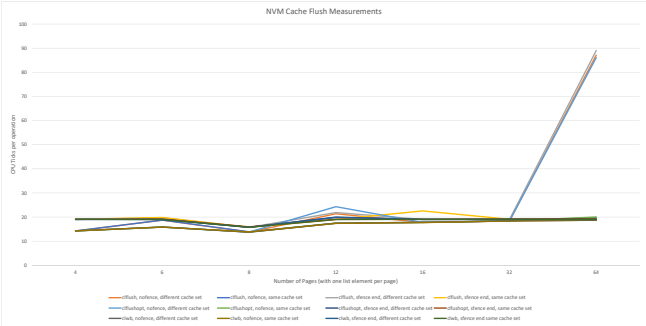
This generic code sets up the "different cache set" and is used to test in the various configurations described throughout this section.

Cache Flush

These tests work by creating linked list entries from a pool of memory pages. Each list entry is at the same offset within the page, which causes them to be assigned to the same CPU associative cache set.

For example, the following code was used for testing clflush without fencing (Figure 4.19):

Figure 4.19: NVM Cache Flush Measurements




```
static unsigned long test_cache_clflush_nofence(record_page_t *rp)
{
    record_t *r = rp->records[0].s.next;
    unsigned long start, end, time;
    unsigned count = 0;

    time = 0;

    while (r != &rp->records[0]) {
        start = _rdtsc();
        r->s.counter++;
        _mm_clflush(r);
        r = r->s.next;
        end = _rdtsc();
        time += end - start;
        count++;
    }

    // report the amount of CPU time
    return time;
```

```
}
```

The goal of this effort was to measure the specific overhead of the given operation: incrementing the counter, flushing the update, and then advancing to the next record in the list. Since the list is a circular doubly linked list, this will access each of the list elements exactly once. The use of the pointers confounds the processor prefetch logic, since it depends upon the value loaded from memory.

The processor tick counter is used for all measurements. This can be converted to time based upon the processor speed but I chose to use ticks for this analysis (the two should be equivalent.)

All of the tests are modeled after similar behavior, where timings are collected around specific operations and attempt to ignore the overhead of the test harness.

The interesting insight in these results is that the **non-flushed** paths experience a substantial penalty once the number of entries in a linked list reach 64 unique entries. This is interesting because the internal memory controller has 56 write back buffers (this is for Skylake Xeon processors).

CLFLUSHOPT

The `clflushopt` instruction was introduced by Intel to provide a mechanism for writing back a single cache line, unlike the earlier `clflush` instruction. This instruction invalidates the given cache line. A subsequent `sfence` guarantees that the cache line is persistent.

This instruction is present on current generation Intel CPUs, but is not present on older CPUs.

Note that `clflushopt` may be executed in parallel with other `clflushopt` instructions within the same instruction stream (e.g., parallel execution is permissible).

Different Cache Set

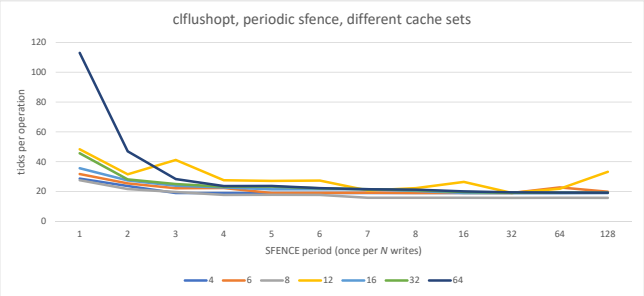
These results are shown in in Figure 4.20.

There is a notable improvement in performance when the sfence operation period is decreased; for this test there is minimal improvement doing it less frequently than once per four write operations.

Same Cache Set

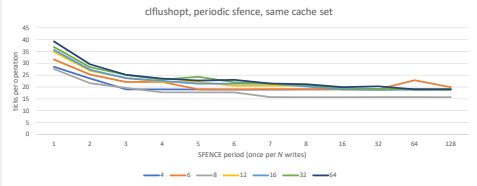
These results are shown in Figure 4.21.

Figure 4.20: NVM CLFLUSHOPT (Different CPU Set)



There is a notable improvement in performance when the sfence operation period is decreased; for this test there is minimal improvement doing it less frequently than once per four write operations.

Figure 4.21: NVM CLFLUSHOPT (Same CPU Set)



CLFUSH

The `clflush` processor instruction on Intel CPUs causes a flush back of the specified cache line. It is distinct from `clflushopt` in that it is serialized with respect to other `clflush` instructions on the same core.

There are suggestions from observed behavior that some implementations of `clflush` may cause flushing of other cache contents as well.

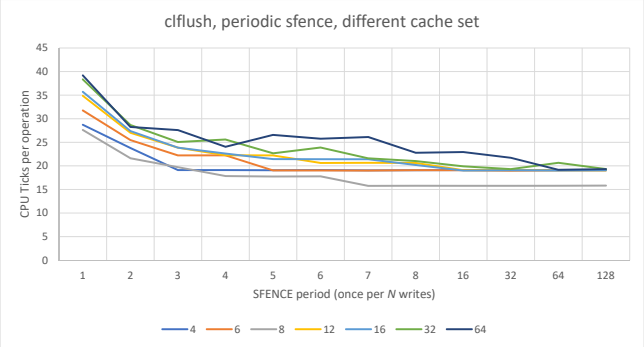
Different Cache Set

These results are shown in Figure 4.22.

The behavior, shown in Figure 4.22, is similar to the behavior observed previously with the `clfushopt` instruction (§4.3.3).

The behavior is not entirely smooth, unlike what I observed earlier with the `clflushopt` instruction (§4.3.3).

Figure 4.22: NVM CLFLUSH (Different CPU Set)



Same Cache Set

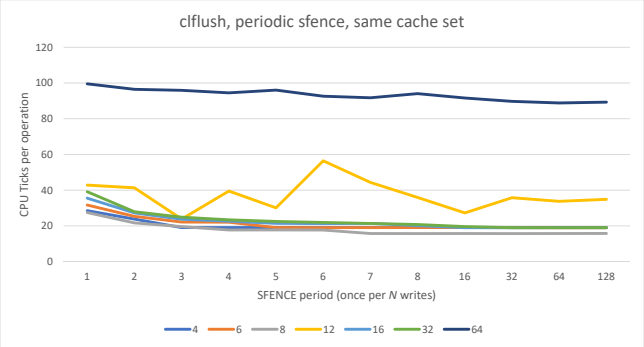
These results are shown in Figure 4.23.

The interesting observation for this test is that it shows an improvement similar to what was seen with `clflushopt` (See §4.3.3). However, unlike the previous result, this one is not as smooth or consistent. This may be a benefit of the parallel nature of the `clflushopt` instruction.

The other interesting departure from the earlier results is that this shows a notable **penalty** for the large size set. The reason for this is not clear from this data; further work may be required.

Finally, when comparing the two different cache set patterns, I observed that the best performance appears to be for the list of size eight. It is not clear to me why this would be the case.

Figure 4.23: NVM CLFLUSH (Same CPU Set)



CLWB

The `clwb` instruction has been recently added in the Intel Skylake Server class CPUs. It enhances the behavior of the previous `clflushopt` instruction by not evicting the cache line from the set — this *writes* data back but does not cause it to be evicted from the cache.

This instruction is useful for data structures that require updating over time, such as counters or various kinds of table structures.

This instruction is expected to be present in future Intel processors.

Different Cache Set

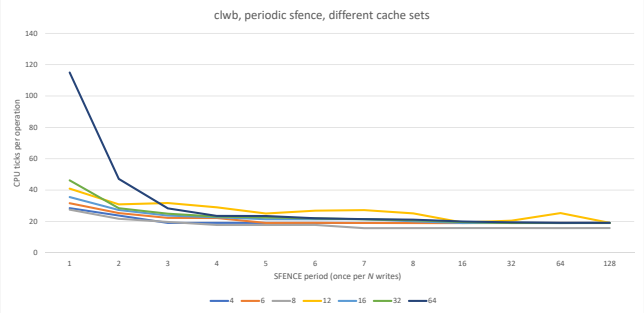
These results are shown in Figure 4.24.

The performance here is similar to that which I observed using the other flush operations — `clflushopt` (§4.3.3) and `clflush` (§4.4).

There are some interesting observations from this data: the 64 entry long linked list set shows very poor performance when combined with aggressive fence operations, but converges towards the mean for less aggressive fence operations.

Much like I previously noted (§4.4), it is interesting that the list of length 8 again has consistently better performance for most periodic fencing operation counts.

Figure 4.24: NVM CLWB (Different CPU Set)

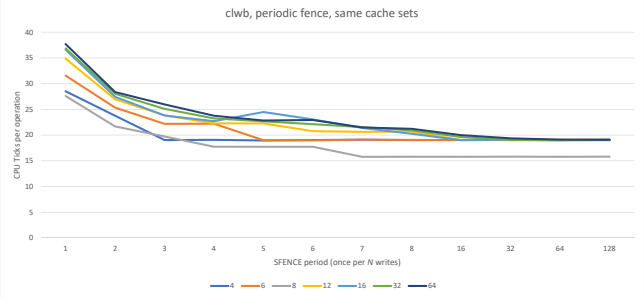


Same Cache Set

These results are shown in Figure 4.25.

Two observations: every linked list set *other than* the eight entry list converges to the same performance (approximately 19 clock ticks). The 8 entry list converges just under 16 ticks. This 20% difference (which I observed previously, see §4.4 and §4.3.3) seems to be fairly consistent throughout the various tests.

Figure 4.25: NVM CLWB (Same CPU Set)



Linked List

The basis for these micro-benchmarks is a circular doubly linked list structure. In this section I have evaluated the performance of those without utilizing cache flushing primitives, though I have incorporated fencing operations in these measurements.

No Flush

The results for the linked list test without flushing is shown in Figure 4.26.

Note that these tests are all from the same cache set.

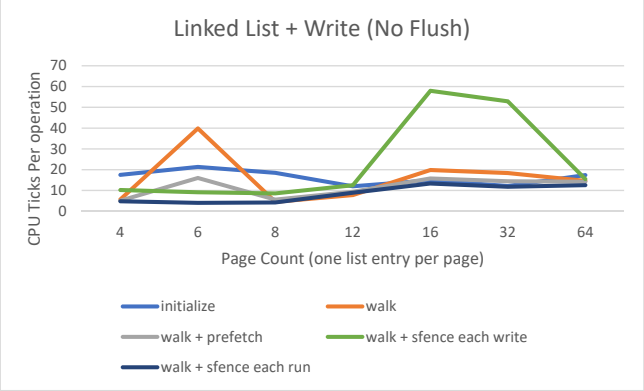
The test looks at five different scenarios:

initialize — this is the performance of the list initialization code. It takes the first 64 byte block of each page given and initializes it to be in a linked list with a zero value for the counter.

walk — this is the cost of walking the list and incrementing the counter.

walk + prefetch — this is where the code prefetches the next pointer value, and walks the list.

Figure 4.26: NVM Linked List Baseline (No Flush)



walk + sfence write — the code in this case walks, but uses the `sfence` instruction after each write.

walk + sfence run — the code in this case walks and updates the counter, performing an `sfence` operation at the end of the entire run (so once per list traversal).

The best overall walk performance came from using an `sfence` operation at the end of each run. The worst walk performance was from using an `sfence` operation after each write.

These were early results and encouraged me to look at further cache/flush behaviors.

Flush

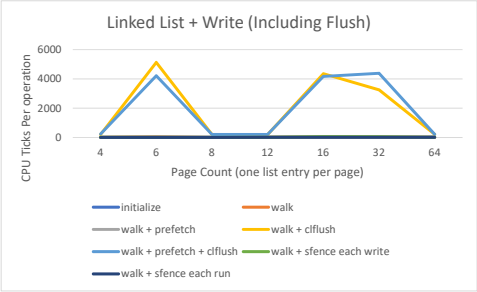
The results for the linked list test with flushing is shown in Figure 4.26.

Note that these tests are all from the same cache set.

In addition to the five scenarios described in §4.6.1 I added two flushing scenarios. The results are shown in Figure 4.27.

The additional scenarios are:

Figure 4.27: NVM Linked List Baseline (With Flush)



walk + clflush — the code walks the list and incrementing the counter, with a `clflush` after each write.

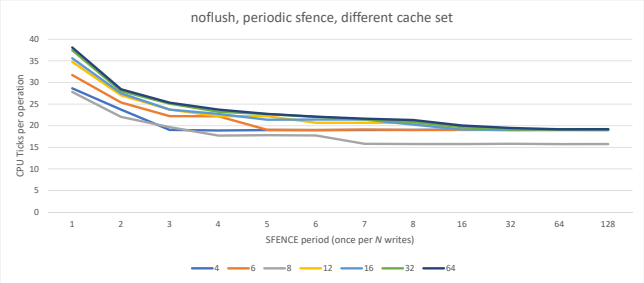
walk + prefetch + clflush — this is where the code prefetches the next pointer value, and walks the list, incrementing the counter, with a `clflush` after each write.

These results surprised me because the cost associated with doing a cache flush seemed to be difficult to predict or control. While all seven tests are represented, the cost associated with the flushing operations causes them to be lost.

Periodic Fence

Evolving from the simple fencing, I broadened my analysis to include periodic fencing. This section describes those measurements. In this case there is no flushing. This helps understand the cost of fence operations.

Figure 4.28: NVM Periodic Fence, No Flush, Different Cache Set



In this subsection I look at the use of periodic fencing with both the same and different cache sets (as described more fully in §4.3.1).

Different Cache Set

The results from these tests are shown in Figure 4.28.

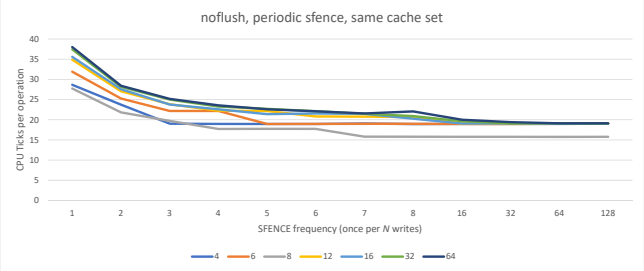
These results are surprisingly similar to those observed when using flush operations (§4.3.3, §4.4, and §4.5). Further analysis may be required to understand if there is any benefit to the cache flush operations.

Same Cache Set

The results from these tests are shown in Figure 4.28.

These result are essentially the same as seen when using a different cache set; this was to such an extent that I went back and confirmed they were not the same — they are not, but they are quite similar.

Figure 4.29: NVM Periodic Fence, No Flush, Same Cache Set



Memory Allocator Results

The memory allocator studies consist of four distinct tests:

Alloc/Free/Alloc Test — this performs a multi-threaded allocate/free/allocate sequence of 100,000 items per thread. See §4.7.1.

Alloc/Free Test — this performs a multi-threaded alloc/free sequence of 100,00 items per thread. See §4.7.2.

Fast Alloc Test — this performs a multi-threaded allocation of 100,000 items per thread. See §4.7.3

Linked List Test — this performs a multi-threaded allocation and insertion into a linked list, 100,000 items per thread. See §4.7.4

For each case the number of threads is varied. All allocations are 64 bytes (fixed).

Five allocators are evaluated. A sixth (Makalu) is not included in these figures because its performance was so poor it made the charts not useful for evaluating the other allocators.

All of these allocators conform to the **malloc** interface. The test framework adds an absolute-to-relative and relative-to-absolute addressing function to enable

storing the pointers persistently as well as accessing them using pointer operations in memory.

The five allocators evaluated are:

malloc — this is one of the baseline allocators and is evaluated using DRAM on the target machine. It uses the system standard `malloc` implementation which on this test system was **glibc**.

jemalloc — this is a NUMA and core aware allocator that provides excellent scalable performance. It is evaluated against DRAM and is the second baseline allocator for this evaluation.

nvm — This is the HPI persistent memory allocator (see §3.3.3 for more details).

pmcto – this is the PMDK experimental allocator with **open/cose** consistency guarantees only — it does not flush or fence.

pmobj — this is the PMDK object allocator with objectallocation guarantees; it is based upon a version of *jemalloc* that has been modified to work with persistent memory.

As I will discuss in §5 additional allocators should be evaluated and/or developed for use with non-volatile memory.

Alloc/Free/Alloc Test

The results for this test are shown in Figure 4.30.

Observations:

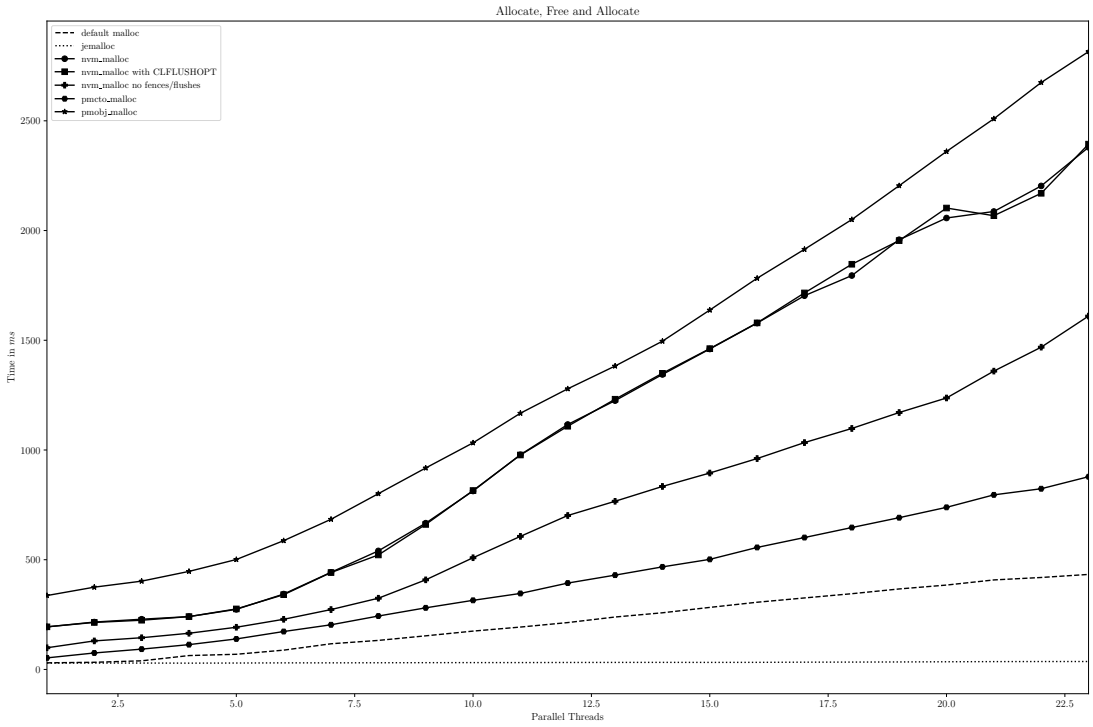
jemalloc — this provides a highly scalable implementation in DRAM. It represent more of a sanity check that the system is working as expected as I would not expect to see performance below this from persistent memory (unless it was *faster* than the DRAM on the system).

malloc — this is not an optimal implementation, but provides an example of a non-NUMA, multi-core optimized allocator can provide.

pmcto — given that this is the optimized PMDK version of malloc without flush or fence, it should represent the optimal possible performance on the current system.

nvm — this result was surprising: this is clearly a better performing allocator than is present in the PMDK. It's performance without flush/fence is not as good

Figure 4.30: Memory Allocation Tests (Alloc-Free-Alloc)



as the PMDK, but it's performance *with* flush/fence is substantially better than the PMDK's object allocator.

pmobj — this is the PMDK's object allocator; it provides variable size objects to the caller, with persistence. Note that the contents of the object are not managed by the PDMK by default; applications must be written to use PMDK interfaces to properly flush and fence.

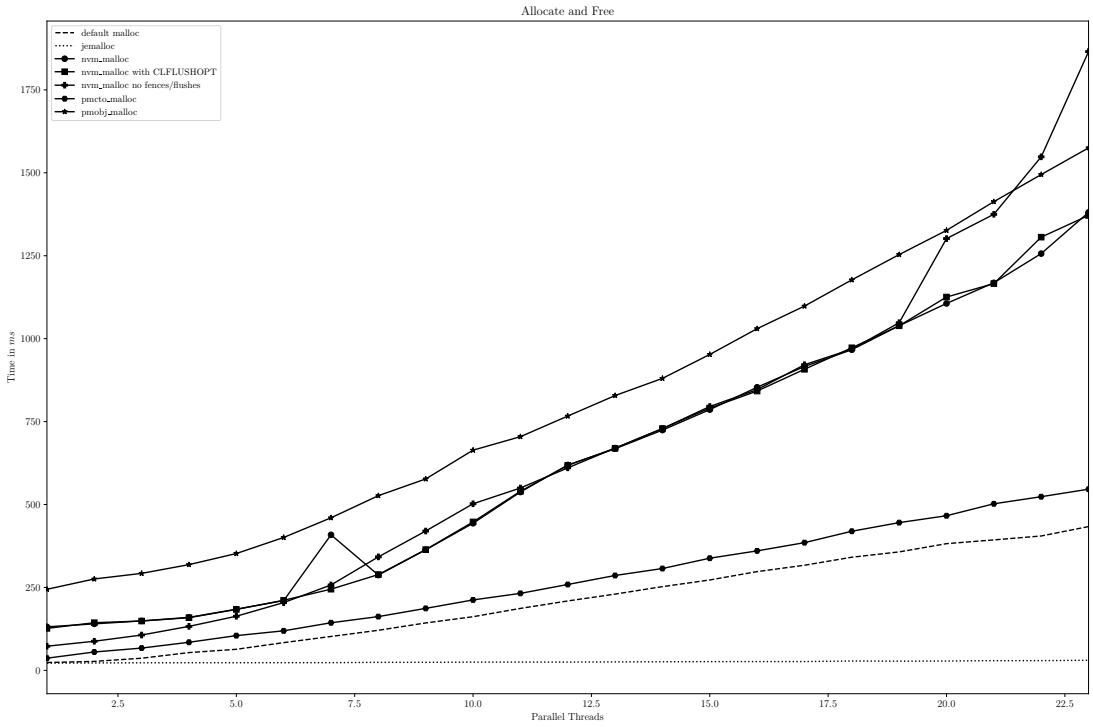
Non-volatile memory allocators are still fairly inefficient. There is certainly room for improvement.

Alloc/Free Test

The results from this test are shown in Figure 4.31.

The observations here are similar to those in §4.7.1. It is interesting to note that once again the HPI allocator ("nvm") has the same performance with or without the use of `clflushopt`.

Figure 4.31: Memory Allocation Tests (Alloc-Free)



Fast Alloc Test

The results for this test are shown in Figure 4.32.

Recall that this is an allocation only test, none of the memory is freed.

These results actually *do* have some surprises in them:

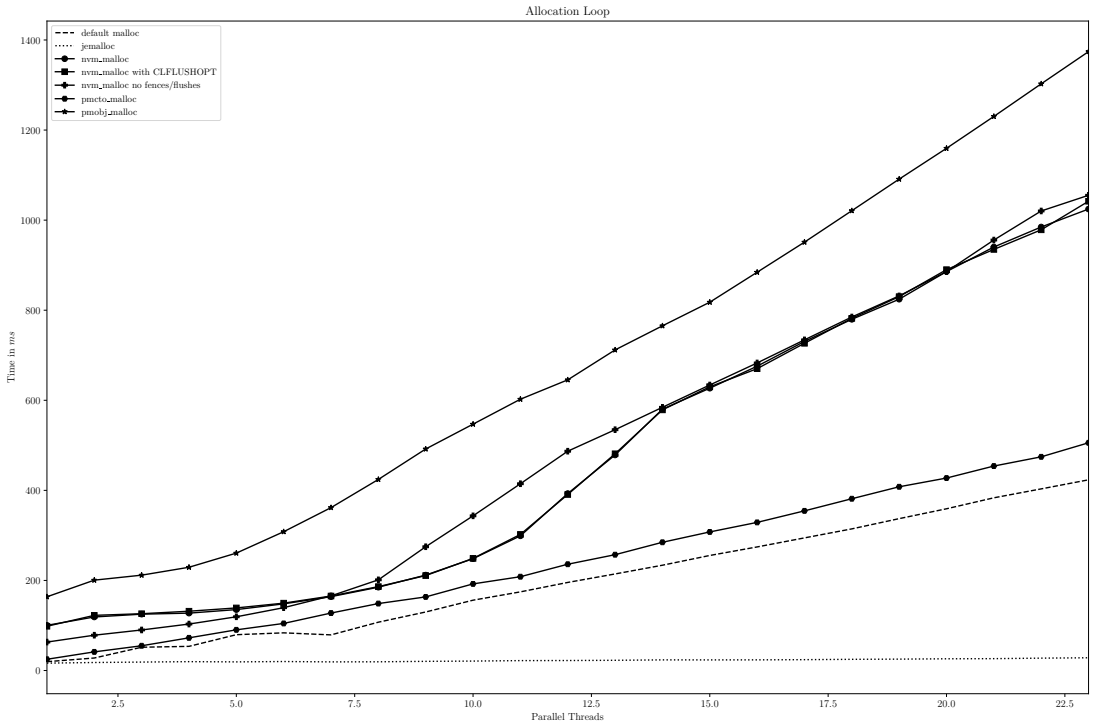
- pmcto is only slightly worse than the glibc `malloc`.
- nvm **without** flush/fence operations performs worse than *with* flush/fence operations on several tests and is remarkably close.
- pmobj remains the least performance allocator shown (Makalu was *far* worse).

This raises interesting questions regarding the efficiency of deletion operations.

Linked List Test

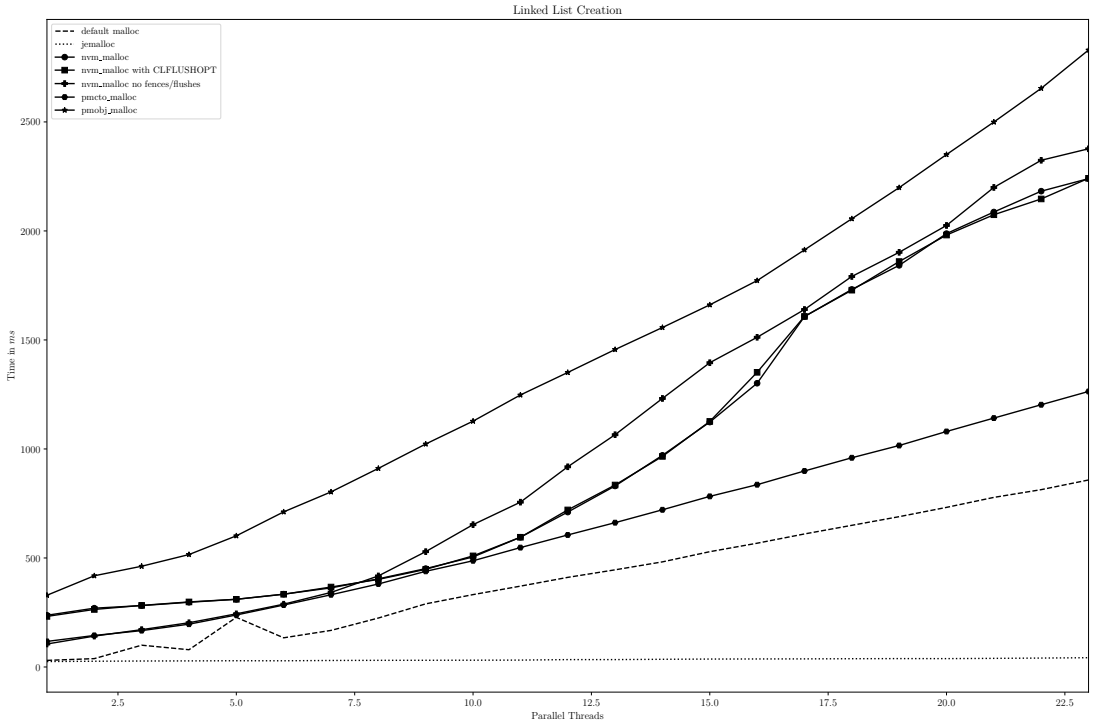
The results for this test are shown in Figure 4.33.

Figure 4.32: Memory Allocation Tests (Fastalloc)



No significant observations from this particular data; may require further analysis.

Figure 4.33: Memory Allocation Tests (Linkedlist)



Chapter 5

Discussion

In this section I attempt to summarize the important observations that have emerged from my analysis and future work that can build upon those findings.

Observations

There are quite a few detailed observations in Chapter 4. My goal is to extract the most significant of those and summarize them here:

- Non-volatile Memory, as currently implemented by Intel and presented in Apache Pass, is not a performance play. This is not a DRAM replacement. To make effective use of it we need to play to its strengths: persistence and size. Techniques such as striping may prove beneficial to improve performance: testing parallel access to *different* NVM modules may provide further insight into whether or not this could be effective.

- Working with non-volatile memory is generally not a "drop-in" replacement. I did not explore this extensively in this research, but the literature fairly clearly points out the problems of mixing volatile and non-volatile memory. In discussing this, one interesting model to help address this was to consider using a portion of the extended address space to present NVM as a form of key-value store. This is discussed (albeit briefly) in §5.2.4.

- Slightly relaxed fencing can lead to substantial improvements in overall performance.

- Cache set awareness can create a natural way to limit the failure window and minimize the cost of performing sfence operations.

- Consideration of failure models is vital to effective use and understanding of NVM. Failure for non-volatile memory requires rethinking how things fail; this is well-described in the literature but easily overlooked. A better model for failure may be useful here.
- Exploiting NVM is likely to require either program language support or purpose build and tuned systems. Program languages can help deal with the problems of mixing persistent and dynamic memory systems. Purpose built systems can be carefully crafted by experienced developers (e.g., systems developers) who are familiar with the challenges of mixing persistent and ephemeral address spaces.

There are quite a few areas that I was not able to explore in the given time period; such is the nature of research.

An open question is how useful it is to explore this research space further, given the present state of the technology and the 25+ *years* that have already been spent analyzing persistent memory.

Future Work

The work that I have done here has raised more questions and potential new avenues to explore than provided answers. Some of these I expect to continue exploring and I describe them here. Some are more speculative ideas that may warrant further exploration. I include both of these to demonstrate some of the fruits of this research project as well as provide a semi-coherent reminder of these thoughts for future consideration.

Transactional Memory

One observation that I have made during the course of this investigation is that *controlling the cache* is one of the challenges in creating persistent memory. Storage developers are used to being able to control their write back behavior; processor caches do not work this way.

Thus, the problem is that ordinary operational events, such as a context switch or interrupts, can cause cache lines to be evicted. This in turn means that atomic updates to memory must be transactional. Many of the schemes described in the literature are *de facto* software transactional memory implementations, whether it is through versioning, logging, or shadowing.

I would suggest exploring the use of hardware transactional memory as a means of managing the cache: it provides a *notification* when partially modified contents in memory have been evicted because the hardware transaction will abort. These

are rare events. I would expect this is a good *special purpose* solution — such as building a high performance persistent storage solution — and could potentially be used by programming language support, but is by itself not a general purpose solution given the pragmatic limitation on the size of transactions.

Thus, future work could evaluate the various transactional memory approaches described here. An investigation into the costs and benefits of implementing common data structures in persistent memory using the various techniques might help construct better systems in the future.

Allocators

Space allocators, such as in dynamic program heaps, slab allocators, buddy allocators, and/or object allocators are known to be challenging to construct for arbitrary usage patterns. Persistent allocation is a complex issue in media file systems, for example, because there is no *a priori* way to optimally allocate space without knowing the usage pattern, which is unlikely to be a common case.

In this report I have evaluated several existing allocators. An area for future work would be to do a more systematic deconstruction and analysis of these allocators to better understand the performance bottlenecks — given a real hardware system, are there ways to improve those existing allocators. Is it possible to construct an allocator optimized for working with non-volatile memory?

To do this I would suggest measuring:

allocation versus free performance — these results show that free is expensive. Why is it expensive? Are there approaches to improving this?

read versus write performance — these preliminary results suggest that the performance of the system is highly dependent upon specific access patterns. How do these impact allocation?

System Benchmarks — these results are all focused micro-benchmarks. Allocators (especially if custom built) should be evaluated within the context of useful systems. Thus, measuring something like *etcd* using existing measurements, for read/write/delete performance and for a range of object sizes. Continue to use micro-benchmarks for improving understanding of the performance profile.

Ultimately the goal of forward-looking work in this area should be:

- Understand the change in relative costs between DRAM and NVM — these directly impact design decisions

- Understand *why* we observe these differences? If they are specific to the hardware, they are less interesting. If they are general to the technology or the problem space, they are more interesting.
- Using these insights try to extract a set of *principles* that provide greater insight into understanding the problems constructing allocators.

One idea, not captured by these previous notes, would be to see if there is any possibility of constructing a log-structured allocator. This led to the discussion of using the extended virtual memory address space as a form of key-value store, as described more in §5.2.4.

Concurrent Persistent Data Structures

The original proposal for this research discussed evaluating persistent data structures. While I gained some low-level understanding of the primitives, the higher level goal of looking at data structure design was largely deferred.

I would suggest that a useful research direction would be to look at how to construct concurrent persistent data structures. To do this one approach would be to consider adding **persistence** to existing concurrent data structures, and independently adding **concurrency** to existing persistent data structures.

Once done, analyzing the lessons learned in doing this and comparing the end state would be useful if it provided greater insight into the tradeoffs inherent in constructing such data structures.

Key-Value Stores

One motivation for evaluating persistent memory was as a potential medium for constructing an in-memory graph file system — the addition of persistence would yield a file system.

An interesting observation that came about while discussing the topics of this research was the observation that one of the challenges in persistent memory is the use of *pointers*. These are problematic because when used in a memory mapped model there are no real guarantees that the same mapping will be subsequently available.

One possible mechanism for handling this was to add an extra level of indirection to the references, so they can be reallocated, such as via compaction. It was at this point that we observed this is essentially the function of a key-value store.

One possible area to research then would be to use a fragment of the virtual address space as a *persistent* part of the address space. For the Intel platform under study, the address space is now 128PB, with the five level page table support, a

single top level entry represents 128TB of address space. If this were used as a form of key-value store, the addresses could be mapped to a valid memory location via the normal page fault handling mechanism.

A more general observation here is that a key-value store optimized for non-volatile memory certainly could present interesting questions: a sort of "NVRAM-Cloud" style project, perhaps.

Chapter 6

Conclusions

The original goals of this project, as set forth in the proposal were to:

Describe a Failure Model — I made some progress in this area, as discussed in §3.1. Thus, this goal has been accomplished at some level.

Durability — much of the micro-benchmark work that I did was exploring the tension between performance and crash durability. Thus, I would note that this goal has been accomplished at some level.

Performance Evaluation — this goal was to evaluate adding non-volatile memory support to those systems. No work was done in this space due to resource constraints. This remains future work.

The **primary objective** of this project was to demonstrate the ability to conduct independent research. While I have worked with a number of people during this project, including discussing preliminary results and possible future directions, the work described in this document rests upon the research that I have conducted. The experimental results in Chapter 4 are directly based upon data that I collected. I wrote the micro-benchmark code, modified the memory allocator benchmark framework, wrote additional scripts to automate running and collecting a vast volume of data.

Thus, I submit to those evaluating my work that this is sufficient to demonstrate the ability to conduct independent research in pursuit of my PhD at The University of British Columbia.

Bibliography

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *computer*, vol. 29, no. 12, pp. 66–76, 1996. → page 11

[2] J. Arulraj and A. Pavlo, "How to build a non-volatile memory database management system," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1753–1758. → page 8

[3] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 707–722. → pages 8, 10

[4] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory." in *HotOS*, vol. 13, 2011, pp. 2–2. → page 9

[5] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to flash memory," *Proceedings of the IEEE*, vol. 91, no. 4, pp. 489–502, 2003. → page 1

[6] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Implications of cpu caching on byte-addressable non-volatile memory programming," *Hewlett-Packard, Tech. Rep. HPL-2012-236*, 2012. → page 9

[7] J. Bøner, "Latency numbers every programmer should know," (accessed September 15, 2018). [Online]. Available: https://gist.github.com/jboner/2841832 → page 2

[8] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 433–452. → page 9

[9] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "Rewind: Recovery write-ahead system for in-memory non-volatile data-structures," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 497–508, 2015. → page 10

[10] A. Chen, "A review of emerging non-volatile memory (nvm) technologies and applications," *Solid-State Electronics*, vol. 125, pp. 25–38, 2016. → page 7

[11] G. Chen, L. Zhang, R. Budhiraja, X. Shen, and Y. Wu, "Efficient support of position independence on non-volatile memory," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 191–203. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124543 → page 8

[12] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using crash hoare logic for certifying the fscq file system," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 18–37. → page 11

[13] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015. → page 7

[14] X. Chen, E. H.-M. Sha, A. Abdullah, Q. Zhuge, L. Wu, C. Yang, and W. Jiang, "Udorn: A design framework of persistent in-memory key-value database for nvm," in *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2017 IEEE 6th*. IEEE, 2017, pp. 1–6. → page 8

[15] D. W. Clay and S. A. Anderson, "Flash solid state drive that emulates a disk drive and stores variable length and fixed lenth data blocks," Oct. 17 1995, uS Patent 5,459,850. → page 7

[16] N. Cohen, M. Friedman, and J. R. Larus, "Efficient logging in non-volatile memory by exploiting coherency protocols," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 67:1–67:24, Oct. 2017. [Online]. Available: http://doi.acm.org.prx.library.gatech.edu/10.1145/3133891 → pages 8, 10

[17] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 133–146. → page 9

[18] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier, "Logic and lattices for distributed programming," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 1. → page 10

[19] F. J. Corbató and V. A. Vyssotsky, "Introduction and overview of the multics system," in *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*. ACM, 1965, pp. 185–196. → page 7

[20] I. Corporation, "Micron and intel announce update to 3d xpoint joint development agreement," June 2018, (accessed September 15, 2018). [Online]. Available: https://newsroom.intel.com/news-releases/micron-intel-announce-update-3d-xpoint-joint-development-program/ → page 4

[21] I. Cutress and B. Tallis, "Intel launches optane dimms up to 512gb: Apache pass is here!" May 2018, (accessed September 15, 2018). [Online]. Available: https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here → page 4

[22] J. Dean, "Software engineering advice from building large-scale distributed systems," (accessed September 15, 2018). [Online]. Available: http://static.googleusercontent.com/media/research.google.com/en/us/people/jeff/stanford-295-talk.pdf → page 2

[23] S. R. Dulloor, "Systems and applications for persistent memory," Ph.D. dissertation, Georgia Institute of Technology, 2015. → page 10

[24] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 15. → page 10

[25] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 2014, pp. 75–88.

[26] R. P. Feynman, "There's plenty of room at the bottom [data storage]," *Journal of microelectromechanical systems*, vol. 1, no. 1, pp. 60–66, 1992. → page 7

[27] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank, "A persistent lock-free queue for non-volatile memory," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018, pp. 28–40. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178490 → page 7

[28] D. Frohman-Bentchkowsky, "Random-access floating gate mos memory array," US Patent 3 728 695, 1973. [Online]. Available: https://patents.google.com/patent/US3728695A → pages 1, 7

[29] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown, "Recon: Verifying file system consistency at runtime," *ACM Transactions on Storage (TOS)*, vol. 8, no. 4, p. 15, 2012. → page 11

[30] E. Giles, K. Doshi, and P. Varman, "Transaction local aliasing in storage class memory," in *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 349–350. → page 10

[31] ——, "Continuous checkpointing of htm transactions in nvm," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. ACM, 2017, pp. 70–81. → page 8

[32] D. Gilmer, T. Rueckes, and L. Cleveland, "Nram: a disruptive carbon-nanotube resistance-change memory," *Nanotechnology*, vol. 29, no. 13, p. 134003, 2018. → page 5

[33] M. Herlihy, "The future of synchronization on multicores: The multicore transformation (ubiquity symposium)," *Ubiquity*, vol. 2014, no. September, p. 1, 2014. → page 10

[34] D. Hitz, J. Lau, and M. A. Malcolm, "File system design for an nfs file server appliance." in *USENIX winter*, vol. 94, 1994. → page 9

[35] A. S. Hoagland, "History of magnetic disk storage based on perpendicular magnetic recording," *IEEE transactions on magnetics*, vol. 39, no. 4, pp. 1871–1875, 2003. → page 7

[36] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 468–482. [Online]. Available: http://doi.acm.org.prx.library.gatech.edu/10.1145/3064176.3064204 → pages 7, 10

[37] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "Lama: Optimized locality-aware memory allocation for key-value cache." in *USENIX Annual Technical Conference*, 2015, pp. 57–69.

[38] K. Huang, J. Zhou, L. Huang, and Y. Shen, "Nvht: An efficient key-value storage library for non-volatile memory," *Journal of Parallel and Distributed Computing*, 2018. → page 8

[39] M. Info, "Stt-mram: Introduction and market status," (accessed September 15, 2018). [Online]. Available: https://www.mram-info.com/stt-mram → page 5

[40] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016. → page 10

[41] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*. IEEE, 2015, pp. 660–671. → pages 7, 10

[42] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 361–372. → page 9

[43] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a true direct-access file system with devfs," in *16th USENIX Conference on File and Storage Technologies*, 2018, p. 241. → page 8

[44] J. Kim, S. Lee, and J. S. Vetter, "Papyruskv: a high-performance parallel key-value store for distributed nvm architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 57. → page 8

[45] W.-H. Kim, J. Seo, J. Kim, and B. Nam, "clfb-tree: Cacheline friendly persistent b-tree for nvram," *ACM Transactions on Storage (TOS)*, vol. 14, no. 1, p. 5, 2018. → pages 7, 10

[46] A. Kolli, "Architecting persistent memory systems," 2017. → pages 7, 11

[47] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 481–493. [Online]. Available: http://doi.acm.org.prx.library.gatech.edu/10.1145/3079856.3080229 → page 7

[48] ——, "Language-level persistency," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 481–493, Jun. 2017. [Online]. Available: http://doi.acm.org.prx.library.gatech.edu/10.1145/3140659.3080229 → page 7

[49] M. A. Lastras-Montaño and K.-T. Cheng, "Resistive random-access memory based on ratioed memristors," *Nature Electronics*, vol. 1, no. 8, p. 466, 2018. → page 5

[50] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "Wort: Write optimal radix tree for persistent memory storage systems." in *FAST*, 2017, pp. 257–270. → pages 7, 10

[51] S. Lee, "Write-optimal radix tree: A deterministic indexing structure for persistent memory storage systems," 2018. → page 7

[52] Y. K. Lee and J. R. Domitrowich, "Regulated mos substrate bias voltage generator for a static random access memory," US Patent 4 322 675, 1982. [Online]. Available: https://patents.google.com/patent/US4322675A → page 1

[53] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE, 2014, pp. 580–591. → page 10

[54] L. Liang, R. Chen, H. Chen, Y. Xia, K. Park, B. Zang, and H. Guan, "A case for virtualizing persistent memory," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 126–140. → page 9

[55] D. Liu, K. Zhong, T. Wang, Y. Wang, Z. Shao, E. H.-M. Sha, and J. Xue, "Durable address translation in pcm-based flash storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 475–490, 2017. → pages 8, 9

[56] H. Liu, L. Huang, Y. Zhu, and Y. Shen, "Librekv: A persistent in-memory key-value store," *IEEE Transactions on Emerging Topics in Computing*, 2017. → page 8

[57] H. Liu, L. Huang, Y. Zhu, S. Zheng, and Y. Shen, "Hmfs: A hybrid in-memory file system with version consistency," *Journal of Parallel and Distributed Computing*, vol. 117, pp. 18–36, 2018. → page 8

[58] G. Lu, Y. J. Nam, and D. H. Du, "Bloomstore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–11.

[59] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*. IEEE, 2014, pp. 216–223. → page 9

[60] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 96:1–96:30, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133920

[61] P. Mahajan, L. Alvisi, M. Dahlin *et al.*, "Consistency, availability, and convergence," *University of Texas at Austin Tech Report*, vol. 11, 2011. → page 10

[62] A. Malventano, "How 3d xpoint phase-change memory works," June 2017, (accessed September 15, 2018). [Online]. Available: https://www.pcper.com/reviews/Editorial/How-3D-XPoint-Phase-Change-Memory-Works → page 4

[63] V. Marathe, A. Mishra, A. Trivedi, Y. Huang, F. Zaghloul, S. Kashyap, M. Seltzer, T. Harris, S. Byan, B. Bridge *et al.*, "Persistent memory transactions," *arXiv preprint arXiv:1804.00701*, 2018. → pages 8, 10

[64] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris, "Persistent memcached: bringing legacy code to byte-addressable persistent memory," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017. → pages 7, 10

[65] F. Masuoka, M. Asano, H. Iwahashi, T. Komuro, and S. Tanaka, "A new flash e 2 prom cell using triple polysilicon technology," in *Electron Devices Meeting, 1984 International*. IEEE, 1984, pp. 464–467. → page 7

[66] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, "Atomic in-place updates for non-volatile main memories with kamino-tx," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 499–512. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064215 → page 7

[67] T. Mikolajick, S. Slesazeck, M. H. Park, and U. Schroeder, "Ferroelectric hafnium oxide for ferroelectric random-access memories and ferroelectric field-effect transistors," *MRS Bulletin*, vol. 43, no. 5, pp. 340–346, 2018. → page 5

[68] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating system support for nvm hybrid main memory," *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS XII)*, 2009. → page 7

[69] G. E. Moore, "Cramming more components onto integrated circuits." *Electronics*, vol. 38, no. 8, pp. 114–117, 1965. → page 2

[70] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*. ACM, 2013, p. 1. → page 9

[71] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 135–148. [Online]. Available: http://doi.acm.org.prx.library.gatech.edu/10.1145/3037697.3037730 → page 7

[72] ——, "An analysis of persistent memory use with whisper," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 2, pp. 135–148, Apr. 2017. [Online]. Available: http://doi.acm.org.prx.library.gatech.edu/10.1145/3093315.3037730

[73] ——, "An analysis of persistent memory use with whisper," *SIGPLAN Not.*, vol. 52, no. 4, pp. 135–148, Apr. 2017. [Online]. Available: http://doi.acm.org.prx.library.gatech.edu/10.1145/3093336.3037730 → page 7

[74] ——, "An analysis of persistent memory use with whisper," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 135–148, Apr. 2017. [Online]. Available: http://doi.acm.org.prx.library.gatech.edu/10.1145/3093337.3037730 → pages 7, 10

[75] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey III, D. R. Chakrabarti, and M. L. Scott, "Dalí: A periodically persistent hash map," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. → pages 7, 10

[76] J. Ou and J. Shu, "Fast and failure-consistent updates of application data in non-volatile main memory file system," in *Mass Storage Systems and Technologies (MSST), 2016 32nd Symposium on*. IEEE, 2016, pp. 1–15. → page 9

[77] I. Oukid and W. Lehner, "Data structure engineering for byte-addressable non-volatile memory," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1759–1764. → pages 7, 10

[78] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm, "Sofort: A hybrid scm-dram storage engine for fast data recovery," in *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, ser. DaMoN '14. New York, NY, USA: ACM, 2014, pp. 8:1–8:7. [Online]. Available: http://doi.acm.org.prx.library.gatech.edu/10.1145/2619228.2619236 → pages 8, 10

[79] I. Oukid, D. Booss, A. Lespinasse, and W. Lehner, "On testing persistent-memory-based software," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*. ACM, 2016, p. 5. → pages 9, 10

[80] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, "Memory management techniques for large-scale persistent-main-memory systems," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1166–1177, Aug. 2017. [Online]. Available: https://doi-org.prx.library.gatech.edu/10.14778/3137628.3137629 → page 7

[81] A. Papagiannis, G. Saloustros, M. Marazakis, and A. Bilas, "Iris: An optimized i/o stack for low-latency storage devices," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 1, pp. 3–11, 2017.

[82] S. Park, T. Kelly, and K. Shen, "Failure-atomic msync (): A simple and efficient mechanism for preserving the integrity of durable data," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 225–238. → page 9

[83] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 265–276. → page 10

[84] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Application crash consistency and performance with ccfs," *ACM Trans. Storage*, vol. 13, no. 3, pp. 19:1–19:29, Sep. 2017. [Online]. Available: http://doi.acm.org.prx.library.gatech.edu/10.1145/3119897 → page 8

[85] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 672–685. → page 10

[86] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory allocation for nvram." in *ADMS@ VLDB*, 2015, pp. 61–72. → page 14

[87] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 91–104, 2017. → page 10

[88] E. H.-M. Sha, W. Jiang, H. Dong, Z. Ma, R. Zhang, X. Chen, and Q. Zhuge, "Towards the design of efficient and consistent index structure with minimal write activities for non-volatile memory," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 432–448, 2018. → page 7

[89] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 323–337. → page 9

[90] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: a flexible and fast software supported hardware logging approach for nvm," in

Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2017, pp. 178–190. → page 8

[91] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang, "Push-button verification of file systems via crash refinement." in *OSDI*, vol. 16, 2016, pp. 1–16. → page 11

[92] L. E. Truesdell, *The development of punch card tabulation in the Bureau of the Census, 1890-1940: with outlines of actual tabulation programs*. USGPO, 1965. → page 7

[93] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory." in *FAST*, vol. 11, 2011, pp. 61–75. → pages 7, 10

[94] V. Viswanathan, "Intel®memory latency checker," March 2018, (accessed September 15, 2018). [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker → page 21

[95] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104. → pages 7, 9

[96] C. Wang, Q. Wei, L. Wu, S. Wang, C. Chen, X. Xiao, J. Yang, M. Xue, and Y. Yang, "Persisting rb-tree into nvm in a consistency perspective," *ACM Transactions on Storage (TOS)*, vol. 14, no. 1, p. 6, 2018. → page 7

[97] T. Wang, S. Sambasivam, Y. Solihin, and J. Tuck, "Hardware supported persistent object address translation," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 800–812. → pages 8, 9

[98] Z. Wang, H. Qian, J. Li, and H. Chen, "Using restricted transactional memory to build a scalable in-memory database," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 26.

[99] Q. Wei, C. Wang, C. Chen, Y. Yang, J. Yang, and M. Xue, "Transactional nvm cache with high performance and crash consistency," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 56. → page 8

[100] M. Wu and W. Zwaenepoel, "envy: a non-volatile, main memory storage system," in *ACM SIGOPS Operating Systems Review*, vol. 28, no. 5. ACM, 1994, pp. 86–97. → pages 7, 9

[101] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan, "Espresso: Brewing java for more non-volatility with non-volatile memory," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 70–83. → page 9

[102] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "Lsm-trie: an lsm-tree-based ultra-large key-value store for small data," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 2015, pp. 71–82.

[103] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang, "Nvmcached: An nvm-based key-value cache," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, 2016, p. 18. → page 8

[104] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories." in *FAST*, 2016, pp. 323–338. → page 10

[105] S. Yu, N. Xiao, M. Deng, F. Liu, and W. Chen, "Redesign the memory allocator for non-volatile main memory," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 49, 2017. → page 7

[106] M. Zhang, K. T. Lam, X. Yao, and C.-L. Wang, "Simpo: A scalable in-memory persistent object framework using nvram for reliable big data computing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 1, p. 7, 2018. → page 7

[107] W.-z. Zhang, K. Lu, M. Luján, X.-p. Wang, and X. Zhou, "Fine-grained checkpoint based on non-volatile memory," *Frontiers of Information Technology & Electronic Engineering*, vol. 18, no. 2, pp. 220–234, 2017. → page 8

[108] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*. IEEE, 2013, pp. 421–432. → page 9

[109] J. Zhou, Y. Shen, S. Li, and L. Huang, "Nvht: an efficient key-value storage library for non-volatile memory," in *Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*. ACM, 2016, pp. 227–236. → page 8

[110] D. Zhu, Y. Li, W. Shen, Z. Zhou, L. Liu, and X. Zhang, "Resistive random access memory and its applications in storage and nonvolatile logic," *Journal of Semiconductors*, vol. 38, no. 7, p. 071002, 2017. → page 5

[111] P. Zuo and Y. Hua, "A write-friendly hashing scheme for non-volatile memory systems," in *Proc. MSST*, 2017. → page 7

[112] ——, "A write-friendly and cache-optimized hashing scheme for non-volatile memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 985–998, 2018. → page 7

[113] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/zuo → page 7