Karlsruhe Institute of Technology

Achieving Optimal Throughput for Persistent Memory with Per-Process Accounting

Master's Thesis
submitted by

Thomas Schmidt

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Jun-Prof. Dr. Christian Wressnegger
Advisor:	Lukas Werling, M. Sc.

27. September 2021 – 12. Mai 2021

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

Karlsruhe, May 12, 2022

Abstract

In recent years non-volatile main memory (NVMM) emerged as a technology for byte-addressable persistent storage accessible similar to DRAM via the CPUs memory bus. Operating systems followed by introducing direct access (DAX) allowing applications to map NVMM into their address space, bypassing the OS on accesses.

We contribute an NVMM usage monitor based on processor event-based sampling (PEBS) capable of detecting accesses to NVMM performed by applications. This can be used to interpolate an application's NVMM utilization and perform system-wide per-process accounting.

Further, prior work has shown Optane DC's throughput to drop with the increasing number of parallel accesses. By limiting the set of schedulable CPU cores for threads that recently accessed NVMM, we also aim to improve the throughput at high numbers of concurrent threads.

Contents

Chapter 1

Introduction

With the recent arrival of commercially available Intel Optane non-volatile main memory (NVMM), its capabilities and limitations are becoming increasingly relevant. NVMM describes a new class of byte-addressable memory that is persistent across power cycles and connected to the processor's memory bus similar to DRAM. [4]

The performance of Optane DC is generally as prior work expected with a lower bandwidth at higher latency compared to DRAM. [43] However, depending on the access pattern, the achievable bandwidth can vary significantly.

For one, while Optance DC is byte-addressable, its underlying media's access granularity is 256-byte. [30] On updates smaller than said access granularity, Optane is required to load the previous 256-byte record, modify its content to then write back the updated record. This causes, small accesses can incur write amplification which lowers the available bandwidth.

Prior work also found Optane DC to drop its total throughput at an increasing number of simultaneous accesses from different threads. [43] Assuming NVMM becomes prevalent in the future, this is a significant issue as more applications may want to adopt the technology.

Further, whilst there are well-known best practices for the usage of NVMM [43], the later issue cannot be solved by individual applications, yet requires a system-wide solution. While previous storage technologies were generally managed as a resource by the operating system, Optane's integration with the memory-bus led to the development of *direct access (DAX)* [**?**]. DAX allows applications to get request the operating system to map data directly into their address space, from which point the OS is no longer involved in the process of persisting data.

Generally, a good idea, as it lowers the overhead on data's path to persistence, the loss of transparency however is an issue on multiple levels. For one the operating system is currently unable to manage or even observe the throughput from and to NVMM for each individual process. Moreover, even detecting whether or

not the mapped NVMM region was used at all is difficult with direct access in use.

In our work, we utilize counter- and sampling-based performance events to regain insight into a process's usage of NVMM. Further, we implement a prototypic application capable of restricting threads that recently utilized NVMM to a subset of the available CPU cores, thereby setting an upper limit on the number of concurrent accesses to NVMM.

The remaining thesis is structured as follows: Chapter 2 describes the relevant background for this work and looks at related works in the field. Chapter 3 dives deeper into the already briefly mentioned issues of Optane DC's limited parallelizability and direct access. We then evaluate the use of performance counters to mitigate the issue in Chapter 4, see how they perform and what we learned from our experience. Afterward, in Chapter 5 we similarly evaluate sampling-based performance counters for the same task, amending our work by implementing core specialization to separate NVMM threads from others. Chapter 6 discusses our contributions and outlines changes Intel could take to help. Finally, Chapter 7 finishes with a conclusion of our work.

Chapter 2

Background and Related Works

This chapter presents the relevant background of commonly used technologies throughout our work as well as related works. We will start with a quick introduction to persistent memory and continue with details on the throughout this work prominently used *Intel Optane DC persistent memory*. Finally, we provide some background on performance counters.

2.1 Persistent Memory

The emergence of memory technologies offering non-volatile, byte-addressable access at a performance level within an order of magnitude of DRAM, such as Intel's 3D XPoint [32], brought upon a new layer in the memory hierarchy of computing systems. This layer is populated by *non-volatile DIMMs (NVDIMMs)* directly attaching to a processor's memory bus, similar to common DRAM. [4,25] In contrast to DRAM, however, data on NVDIMMs persists over power cycles of the system. Furthermore, it supports a load and store programming model using the respective CPU instructions, unlike typical block-device storage [7], supporting access to storage at lower overheads.

2.1.1 Direct Access Mode

Applications can either use system calls like *read* [22] and *write* [24] to selectively access a files content or *mmap* [19] to get the file's entire contents mapped into their address space. While typically, the operating system's *page cache* is involved in reading and writing to files and mapping pages for the use with *mmap* [12], this is not necessary for NVMM.

Common block storage that is not instantiated with *direct access mode (DAX)* will trigger a page fault on an application's reading accesses when data is not

available in the page cache, then loading it from the device. The page cache maps data sectors from block devices to memory which allows byte-granular access by the processor in contrast to operating at sector- or even memory-page size.

By design, the attributes of NVMM meet those requirements, which resulted in operating systems adapting *direct access (DAX)* [12, 26]. In addition to that, files can be mapped directly into the userspace of a DAX-aware process, and filesystem [26], avoiding the overhead of additional copies on the standard block I/O path. Due to the efficiency gains of removing unnecessary copies, processes are even encouraged to use DAX.

Consequentially, after initially mapping the memory region, accesses onto NVMM performed by a process are opaque to the operating system. Furthermore, as a processor's access to memory via load and store instructions is synchronous, latency and throughput significantly impact the overall system performance. To avoid possible CPU stalls, it is fundamental to have low latencies and high throughput.

Wang et al., for example, used the read and write system calls in their work about *NUMA-Aware Thread Migration for High Performance NVMM File Systems* [40] to measure the amount of data a thread interacted with. As it is reasonable to use Optane DC in combination with DAX, the operating system, however, is unable to make a similar association for an application's access to mapped data. With DAX, accesses in reading and writing directions will utilize load/store semantics that are effectively opaque to the operating system and its page cache.

2.2 Intel Optane

As currently the only widely available persistent memory module, *Intel Optane DC Persistent Memory* offers data persistence at high capacities of up to 512GB per DIMM. [3, 4] These NVDIMMs are compatible with *2nd Generation Intel Xeon Scalable ("Cascade Lake")* and newer processors and can be configured in a variety of configurations. Among other things, they can be used as *volatile memory* in a so-called *Memory Mode*, whereby they serve as the system's main memory and use neighboring DRAM DIMMs on the same channel as cache. Secondly, configured as *persistent storage* in *App Direct Mode*, Optane DC can be presented to the system as a common block device. [4]

We will focus on exploring Intel Optane used as persistent storage for this work. The *memory mode*-usage scenario is not relevant to our work.

2.2.1 Performance Characteristics

While the overall performance characteristics are promising, it was believed that Optane NVMM would behave similarly to DRAM. Yet, the reality is more nuanced with overall lower performance, according to prior works by Yang et al. [43] and Peng et al. [30]. For further reading, we also recommend Wang et al. [41] with their work modeling Intel Optane DIMM for a better understanding of the discrepant performance characteristics.

Access Size While Optane Memory is byte-addressable, the underlying media currently uses chips with an access granularity of 256-bytes [41], which causes the performance to vary depending on the access pattern. This larger access granularity of the media results in generally better performance for sequential accesses [30]. In case of a write smaller than 256-bytes, overhead incurs, as the media will first need to read the 256-bytes, modify its content to write it back afterward [41]. This overhead is called *write amplification* and is often than into account when designing applications using NVMM [35]. This is also highlighted by the fact that the NVDIMM can buffer and merge writes, which helps to reduce the overhead for writes. [43]

Latency As for reads, the latency of Optane Memory is about twice to three times as high in comparison to DRAM and is shown to be more sensitive to data locality. [30,43] Evaluating stores, one finds the latency of DRAM and NVMM is similar to one another and generally consistent. [30]

Bandwidth The maximum bandwidth achievable by a single Optane Memory DIMM is about 6.6 GB/s when reading and 2.3 GB/s writing. Thus, the maximum bandwidth is overall lower than DRAM and shows a significantly higher *read-write gap*. Whereas DRAM has a small gap of about *1.3x* between read and write bandwidth, the same metric for Optane is *3.3x* [30]. [43]

A significant influence on the write bandwidth can be observed depending on whether temporal or *non-temporal* stores are used by the application. Whereas temporal stores are passed along the cache hierarchy of the processor and written back to Optane on a cache line evict or write back (*clflush, clwb*), non-temporal stores are directly written to the persistent memory. The latter shows an overall higher throughput at a lower latency [43]

In a scenario with multiple Optane DIMMs, one can use interleaving to scale the bandwidth for reading and writing almost linearly. [43]

Parallelism Generally, parallelism for Optane DC is not as well explored as the above listed. With increasing numbers of threads accessing Optane DC, Yang et al. [43] found that between one and four threads, performance for reading and writing will peak, after which it will diminish.

Interleaving multiple DIMMs can be of help here, too, yet does not seem to scale linearly with the amount used. One of the authors recommended principles for building software for Optane-based systems even states to "Limit the number of concurrent threads accessing an Optane DIMM" [43].

This has led other researchers like Yang et al. to fall back on dedicated I/O threads for their file system *SPMFS* [44]. They implement a custom I/O handling framework in combination with SPMFS that builds upon an independently managed thread pool to avoid resource contention on the DIMM.

2.2.2 Performance Metrics

Each Optane NVDIMM has its own health and performance metrics that can, for instance, be accessed using *ipmctl* [37] or derived tools like *pmwatch* [8]. Most prior work only used *pmwatch* [8] to evaluate their works, with [14, 45] being examples of this. NThread [40] on the other hand even incorporated it into their designed solution.

Among various others, Optane NVDIMMs promote performance metrics such as the number of load/store operations received from the CPU's integrated memory controller as well as the number of load/store operations performed to the physical media of the NVDIMM [8]. As stated in Section 2.2.1, the access granularity to the underlying media is at 256-byte larger than the desired byte-granularity from the CPU and could cause write amplification for smaller accesses.

While both metrics can be used to measure the bandwidth of the NVDIMM to the CPU, they are not sufficient to account for the usage of each process. As those are measured after the memory controller, changes in these metrics cannot be precisely correlated to a single, respective origin. A possible cause may be a remote NUMA access, but even parallel accesses from multiple local cores cannot be differentiated. Restricting access to a single NVDIMM to only one core at a time is unfeasible, as it would reduce the achievable bandwidth and thus lower system throughput.

2.3 Thread Affinity

Most application processors nowadays are general-purpose, and thus processes and their respective threads can run on all available cores. However, carefully se-

lecting a thread's core can provide performance benefits for the task at hand and the entire system. The most prominent example of this is *cache affinity*, as operating system schedulers try to rerun threads on the same core to reuse accumulated data and instructions of the lower level caches. [39]

Respective APIs to restrict a thread to use a set of cores are found in the kernel and the userspace. [15]

2.3.1 Core Specialization

A different approach is outlined by Gottschlag et al. in "Automatic Core Specialization for AVX-512 Applications" [6]. During AVX-512 instructions, Intel CPUs will reduce their frequency, impacting the overall system performance, as a sibling hyperthread is also affected by this, and restoration of the higher frequency happens delayed. Their work detects threads using AVX-512 instructions and restricts them to a subset of all available cores to reduce the overall impact on performance.

NThread As for NVMM, a paper from Wang et al. [40] describes a mechanism to reduce the number of remote NUMA accesses by threads of an NVMM-based filesystem. As the filesystems currently do not consider NUMA, threads may start by default on a suboptimal core concerning their NUMA node for the files they serve. NThread, by default, migrates the threads so that more local NUMA accesses can be performed, yet also applies an analysis to detect resource contentions. Contention on the NVMM is detected by assuming a maximum bandwidth for Optane DIMMs and using DIMM performance counters to measure their current bandwidth. Once contention is detected, threads with a high write ratio are displaced to different CPU cores where remote NUMA accesses are required. However, the authors mention that their technology does not support *mmap*-ed files. [40] We assume this is because there is no easy way to associate the bandwidth usage of any process due to synchronous memory accesses bypassing the operating system, more specifically the page cache.

2.4 Performance Monitoring Units

The performance of modern processors can typically be evaluated using embedded *performance monitoring units (PMU)* that measure different *performance parameters*, such as instruction cycle, cache hits or misses, and various others.

In the case of Intel processors, each logical processor has its own set of general=purpose performance counters. [11, Vol. 3B, Ch. 19.2.1] Each logical pro-

cessors' performance parameters can be acquired by configuring the appropriate *model-specific registers* with a desired *performance monitoring event (PME)*.

While some PMEs are *architectual*, making them more universally available, others are *non-architectual* and specific to a processors' microarchitecture. The most prominent non-architectural events measure off-core subsystems (so-called *uncore*), which among other things includes the integrated memory controller (iMC) [10]. [11, Vol. 3B, Ch. 19.3.1] As those subsystems are shared among all cores on the same physical die [10], the measurements cannot be directly correlated to an individual core without additional effort, if at all. However, there is a slightly different mechanism to collect *offcore events* allowing the measurement of events that are caused by a processor's core yet leave it for the uncore. [42]

Further details, as well as a reference of the events supported by each Intel CPU, can be found in the Software Developer's Manual Volume 3 in "Performance Monitoring" [11, Vol 3B, Ch. 19].

2.4.1 Processor Event-Based Sampling

While most performance monitoring events provide the developer with counters, some can also be used in connection with *processor event-based sampling (PEBS)* for saving precise architectural information, such as the state of the general-purpose registers and the instruction pointer. PEBS events will be collected after an associated counter reaches a selected *sampling period* and written into a records buffer. Once the buffer is close to being filled, an interrupt is generated to collect the records. [11, Vol. 3B, Ch. 19.6.3.8]

The idea of utilizing PEBS to sample memory accesses to NVMM is already used by Oh et al. in their paper about *MaPHeA*, a profile-guided heap allocation framework with low overhead. Their intention is to reduce the average memory access latency with knowledge about the memory hierarchy. Often used data structures may be allocated on faster memory such as DRAM, whereas less often used ones may be allocated to NVMM. Another interesting contribution of theirs to our work is the insight into the significantly stagnating benefit of high sampling frequencies (or low sampling periods). [28]

Chapter 3

Problem Analysis

As persistent memory is a relatively recent technology, a large part of prior works predates its commercial availability. Researchers until now founded their work on assumptions as to how they expect NVMM will perform, emulating it from their point of view, fitting ways, and performing a variety of studies on these [1, 2, 5]. It was widely expected that Optane DC would have higher latency and lower bandwidth than DRAM. Yet, researchers also find its performance highly dependent on the number of concurrent threads accessing it. [43] Moreover, the access pattern also has a significant impact on its performance.

In this chapter, we first describe how the achievable performance depends on the type of access for Optane DC. Next, we explore the problem of limited parallelism and finish by detailing the issue of detecting utilization in direct access mode.

3.1 Dependence on Access Type

Prior research shows that Optane DC's performance depends on the kind of access (i.e., reads and writes) and their pattern. [43] While the available bandwidth for writes is already limited compared to reads, the problem amplifies for small updates due to the media's physical access granularity of 256-bytes. [41] On each update smaller than said access granularity, the current contents must be read, updated, and written back to the media.

Considering that the granularity is larger than cache lines at 64-bytes, the NVDIMM was found to have a small buffer, sometimes referred to as *XPBuffer* for combining adjacent writes. [43] Yet, random accesses smaller than 256-bytes without much locality cannot meaningfully benefit. A small, random access pattern causes congestion on the buffer and leads to it needing to flush back to the media more often. The required read, update, and the following store increases the

so-called write amplification, which negatively impacts the achievable throughput of the NVDIMM significantly.

The above motivates taking a closer look at stores and what could be done to reduce congestion of the buffer, improving the write throughput of the NVMM.

Finally, it is the reason for our work focusing solely on the evaluation of writes to NVMM. Discovered mechanisms can potentially be used analogously in reading direction with minor adaptions. Nevertheless, we acknowledge that reads should be evaluated in future work, as the read bandwidth is still significantly lower than what is achievable with DRAM.

3.2 Limited Parallelism

Optane DC's performance is dependent on the number of parallel accesses performed. Whereas the observable bandwidth for DRAM increases monotonically for reads and writes with an increasing number of threads accessing it in parallel, Optane was observed to peak at a low number of parallel accesses of one to four threads. [43] One of the reasons is that with an increasing number of threads storing, the contention on the *XPBuffer* increases, as each thread will likely have a separate working set. Because of this, its effectiveness in mitigating write amplification decreases.

This limited ability to parallelize leads to related works like *SPMFS* [44] introducing dedicated I/O threads for their file system. However, in real-world scenarios, an individual application of NVMM cannot control this issue on their own. More and more applications will likely support NVMM, thus requiring coordination among applications to avoid congestion.

However, we believe it is worth considering having the operating system be part of a solution by adapting the scheduling based on the NVMM usage pattern.

3.3 Detecting Direct Access

Due to the limited bandwidth and performance specifics at different access patterns, applications need to be far more mindful of their use of Optane DC than of DRAM. Deploying Optane DC as persistent memory, applications can access it using read/write system calls but are encouraged to use *direct access* (see 2.1.1) for improved performance and reduced overhead. While the throughput can be well observed for system-call-based accesses, once an NVMM-based memory region is mapped into an application's address space, all accesses happen synchronously via load/store instructions of the CPU, transparent to the operating system.

Thus, a supervising system is unable to account for the NVMM usage of an application, let alone able to detect if it was used at all. This issue is commonly known and resulted in papers, such as *NThread* [40], being unable to support an *mmapped* usage scenario.

We believe a mechanism to detect those accesses is crucial when optimizing the usage of NVMM with software, as one is missing a significant part of the picture otherwise.

Chapter 4

Performance Counter for Usage Estimation

As established, we seek to find a way to reliably detect the usage of NVMM in processes and prefer associating NVDIMM usage with their causing processes. Current solutions cannot detect and account for accesses to regions mapped into the address space of a process. We have already seen that the performance metrics provided by the NVDIMM itself are insufficient for our needs due to the inability to associate changes in the metrics with a CPU's cores.

Based on the knowledge that we need to utilize a source of information closer to the CPU cores, we evaluated the use of *performance monitoring units*. Their general availability and low overhead in counting mode, especially if only a few events would be sufficient, make them ideal for continuous monitoring. [27] The ability to acquire them transparently to an application by the kernel or a separate monitoring application using the *perf API* [13], such as the respective binary application *perf* itself, is an essential requirement for unbiased results.

Nevertheless, we believe this approach is still limited in that the PMEs cannot capture the size of the individual accesses of each process. However, the number of accesses per time slice can still be a useful measure to compare different processes or the same process over time.

In the following, we first introduce how we planned on using *performance monitoring units* to achieve per-process accounting. We continue with how we decided on the events we planned to use and show preliminary results from testing. Finally, we conclude with what we have learned from this approach.

4.1 Design

We set out to use PMUs for per-process accounting. Assuming appropriate events are chosen, they are influenced only by the retired instructions of the observed CPU core and resistant to outside factors. Moreover, as those provide an accurate picture formed by the instructions executed, we assume a high level of accuracy in detecting writes. An implementation into an operating system's kernel would also provide a mapping of currently executed processes to their running cores, making it possible to correlate the measured events to the processes.

Our solution configures a limited set of performance counters per CPU core to the relevant events acquiring information about the current NVMM usage. We prefer to set as few counters as possible and evaluate if a single event may be sufficient.

With our PMUs set up with the desired events, before the scheduler switches the context over to the process to be scheduled, we read the value of the current counter. Later, we compare the previously read value with the updated one once we return from the process's context to the scheduler, be it for an interrupt or as the time slice is exhausted. The difference in the before and after is then used to determine how often the process accessed NVMM.

The number of writes triggered does not indicate the amount of data written, as stores can vary in size. Given the granularity for writing to the media of 256 bytes, we can determine an upper limit for an absolute bandwidth measurement. Nevertheless, our design can be used to relatively compare the NVMM usage of different processes in the same time frame or a single process over time. We believe while absolute measures may provide better results, a relative comparison between processes is a good start for bandwidth measurement.

4.2 Event Selection

Support for Optane DC persistent memory is a relatively new feature and requires the *Cascade Lake* architecture or more recent. As our later evaluation happens on a CPU based on *Cascade Lake*, we focused our event selection on said architecture. A list of all supported performance monitoring events can be found in human-readable [9] as well as machine-readable form [29].

In our search for appropriate PMEs, we limit ourselves to *on-core events*, as in particular *offcore* events can be triggered not only by the currently observed core but also by its siblings and remote NUMA accesses. Thus, we encounter the same issues already observed with the performance metrics of the Optane NVDIMM itself.

Next, as we focus this work on writes, we identify a subset of the PMEs that

can be used to detect writes to the NVDIMM. This results in the list of appropriate events found in Table 4.1. Those have in common that they are increased on each *request of ownership (RFO)* of the core for a cache line, which is typically triggered before a write can occur.

While most of these still describe specific scenarios, such as an RFOs before prefetching for *L2* or *L3* cache levels and multiple *bus snooping* configurations, the event *OCR.ALL_RFO.PMM_HIT_LOCAL_PMM.ANY_SNOOP* in particular seems, based on its name, to be summarizing all others. However, we assume prefetching causes additional RFOs that may not occur, due to its speculative nature.

OCR.ALL_RFO.PMM_HIT_LOCAL_PMM.ANY_SNOOP
OCR.ALL_RFO.PMM_HIT_LOCAL_PMM.SNOOP_NONE
OCR.ALL_RFO.PMM_HIT_LOCAL_PMM.SNOOP_NOT_NEEDED
OCR.ALL_PF_RFO.PMM_HIT_LOCAL_PMM.ANY_SNOOP
OCR.ALL_PF_RFO.PMM_HIT_LOCAL_PMM.SNOOP_NONE
OCR.ALL_PF_RFO.PMM_HIT_LOCAL_PMM.SNOOP_NOT_NEEDED
OCR.DEMAND_RFO.PMM_HIT_LOCAL_PMM.ANY_SNOOP
OCR.DEMAND_RFO.PMM_HIT_LOCAL_PMM.SNOOP_NOT_NEEDED
OCR.PF_L2_RFO.PMM_HIT_LOCAL_PMM.ANY_SNOOP
OCR.PF_L2_RFO.PMM_HIT_LOCAL_PMM.SNOOP_NOT_NEEDED
OCR.PF_L3_RFO.PMM_HIT_LOCAL_PMM.ANY_SNOOP
OCR.PF_L3_RFO.PMM_HIT_LOCAL_PMM.SNOOP_NONE
OCR.PF_L3_RFO.PMM_HIT_LOCAL_PMM.SNOOP_NOT_NEEDED

Table 4.1: The list of possible performance monitoring event candidates for Cascade Lake to detect writes for mapped NVMM regions.

4.3 Evaluation

With our set of performance monitoring events defined, we set out to test our hypothesis of those events being a good fit for detecting write accesses to NVMM after which we interpret the results.

4.3.1 Testing Methodology

All benchmarks in this work will utilize our custom *pmm-writer* load generator, which tightly loops around configurable store instructions. These instructions can be temporal or non-temporal and of different sizes.

For our testing, we opted to run *perf stat -e <event>* [13] for all events listed in 4.1 against our *pmm-writer* in different configurations. To ensure stable results, we used *taskset* to ensure the benchmark was consistently scheduled on the same CPU core. We ensured the selected to be part of the same NUMA group as the NVDIMM.

Each benchmark writes a total of 512 MB into a 256 MB large file. Aside from this, each iteration runs at different access sizes, ranging from 1 byte to 256 bytes, with sequential or random write destinations. Finally, we also verified the behavior of temporal and non-temporal stores.

To reduce fluctuations, each configuration was run three times, and their results were averaged for the below results.

4.3.2 Results

While most of our selected PMEs provided us with non-zero counter values during our runs, seven of them did not give any information and thus will not be further evaluated. Those events include all *SNOOP_NOT_NEEDED* variants, as well as the events counting pre-fetches for requests for ownership on the L3 cache, did not provide any results.

For the remaining events, we continued our evaluation by calculating the ratio of reconstructed usage u_r we were able to detect based on our known access sizes s and the final performance counter value c after writing U_t bytes, with U_t as previously mentioned being 512 MB using the following formula:

$$Ratio of Reconstructed Usage u_r = \frac{CounterValue c * FixedAccessSizes}{TotalUsage U_t}$$

In the best case, promising high detection rates and thus overall accuracy will be achieved when u_r is closest to 1. A lower score can be interpreted as an underestimation of the actual usage, whereas a higher score than one highlights an overestimation.

Temporal Writes

Out of our remaining performance monitoring events, we found that the events responsible for counting RFOs before prefetching on the L3 cache and the overall summarizing prefetching-only counters are inconsistently able to provide us with insight. Often they will turn out to provide a counter result of zero, even for multiple runs. Considering the specialized nature of those events and that prefetching is only an optimization that is not always performed beforehand, those results are unsurprising.

Outliers at 256-bytes access size The Figures 4.1 and 4.2 show the usage ratio as defined in Section 4.3.2 for each PME at different numbers of threads. A value of 1 represents a perfect reconstruction of the data usage, whereas a low value shows underestimation and vice versa.

Looking at these figures, one will see significant outliers for the access size of 256-bytes, which are due to our calculation assuming accesses to be of the access sizes s and not being split up into multiple accesses. While the CPU supports vector instructions that can manipulate data more extensively than 64 bytes, the memory bus itself is designed with cache-line size in mind. In addition, when accessing NVMM using vector instructions, prominent examples such as PMDK's *libpmem* will first align their accesses by cache-line and only then use the largest possible instructions for each access. [31] We believe this improves the performance, as partial updates will require reading the existing data and updating them to write back the new contents, which is a potentially avoidable step. Further, this may help to reduce write amplification.

However, as our model assumes each store is indifferent to its size and results in one increment of the performance counter, the 256-byte accesses are overrepresented. Theoretically, they show four times higher reconstructed usage ratio than other access sizes in our graphs. As an example, the actual performance counters values show an average of *7859510 accesses* for demand-based temporal stores from runs with 256-byte access size, compared to *7753884 accesses* for the same with runs of 64-byte access size. Thus they are very similar, with 256-byte detecting 1.3% more write attempts. We will thus concentrate on the smaller access sizes for the remainder of this section.

Next, we want to take a closer at the *demand RFOs* up to and including the 64-byte access size. We observe performance counter values resulting in a reconstructed usage from *67% up to 100%*, with no correlation between the detected usage and their access size. Still, random accesses are performing slightly better with the lowest accuracy of *73%* of the total written data detected. Moreover, the *any snooping* variation of the events performs consistently better across sequential and random access modes.

For the accumulating *ALL_RFO*-events, we find a generally identical picture, which is unsurprising considering most usages were already detected by the herein included *DEMAND_RFO*'s. Taking the prefetching counters into account still increased the lowest calculated reconstructed usage to *97%*, at the expense of an overshoot of *33%* of the written data, up to *70%* for an outlier for random write accesses.

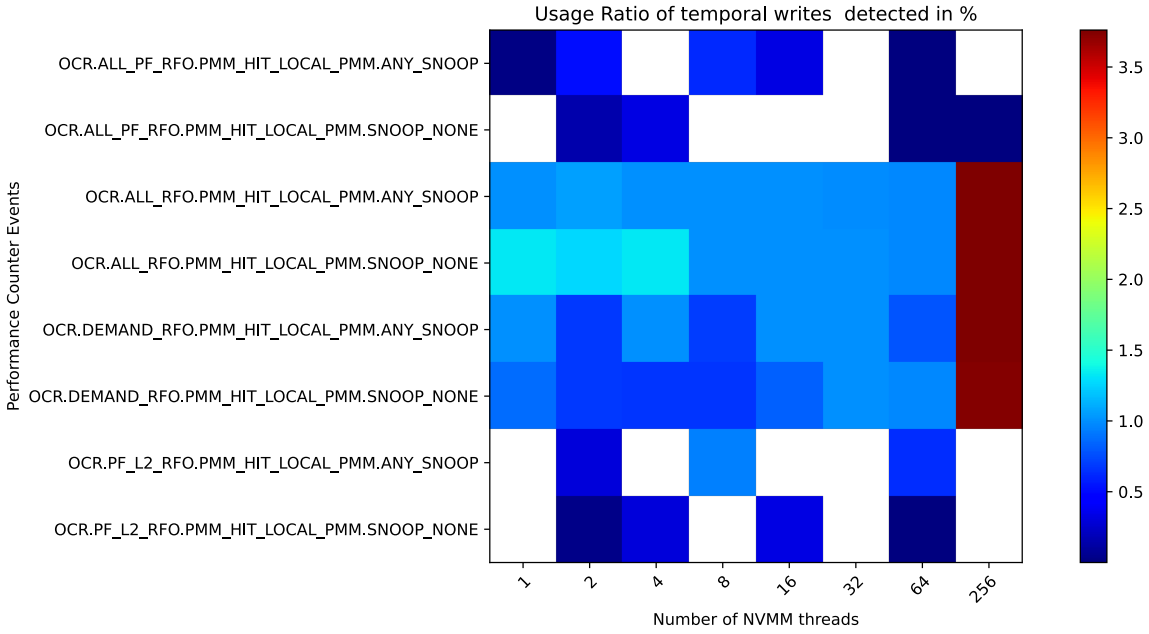

Figure 4.1: Relative detected usage ratio for sequential temporal writes per PME at different access sizes. *DEMAND_RFO* as well as *ALL_RFO* related events seem to be the most promising. A white field is the result of the PMEs counter returning zero.

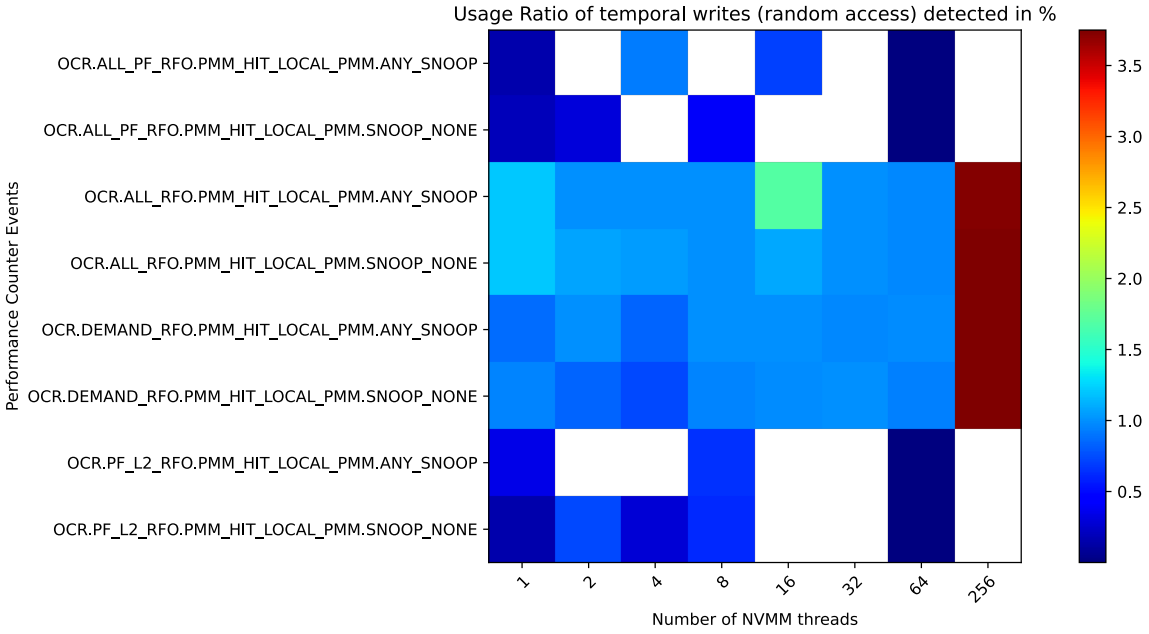

Figure 4.2: Relative detected usage ratio for temporal writes with random access pattern per PME at different access sizes. In comparison to temporal writes, we detect overestimation of the reconstructed throguhput compared to the actual usage of up to 70%. A white field is the result of the PMEs counter returning zero.

Non-Temporal Writes

Looking at the sole prefetching-related events, we find a similar picture in Figure 4.3 and 4.4 as already discussed in connection to the temporal stores. Those are not consistently providing reliable insight to reconstruct usage on their own, which, again, was to be expected due to their limited responsibility.

Full Cache-line Aligned Writes An important discovery made during our benchmark is the significantly limited ability to detect non-temporal stores to NVMM. During our runs with an access size of 64-byte or 256-byte, as previously discussed, we would theoretically expect about *8388608 detected accesses* to fully reconstruct the usage. In practice, we however saw *62 accesses* at the highest, *11,375 accesses* on average with a median at *5 accesses* and thus are in comparison neglectable. We believe this to be caused by non-temporal stores not requiring an RFO to be storable without conflict, but a broadcasted invalidation to be sufficient to achieve cache consistency.

Similar to temporal stores, we can see *DEMAND_RFO*-based events to at first slightly underestimate by up to *3%*. Genereally we find that the reconstructable usage ratio u_r for *DEMAND_RFO*-based PMEs decreases exponentially with increasing access size s and can be well approximated by $u_r = 1 - \frac{s}{64}$. The *SNOOP_NONE*-variant has a significant outlier for sequential at the access size of 2-bytes, For random accesses, a similar outlier can be observed for the *ANY_-SNOOP*-variant at one, and two bytes access size, both at about *68%* instead of the estimated value of over *96%*.

Also similar to temporal stores, *ALL_RFO*-based events generally overestimate the reconstructed usage, potentially by as much as 54%. Nevertheless, we again find the above formula to be a good approximation of the reconstructable usage. The significant outliers originate from *OCR.ALL_RFO.PMM_HIT_LOCAL_PMM._SNOOP_NONE* and focus on the access sizes of 8-byte as well as 16-byte.

4.4 Conclusion

Based on the above results, the selected performance monitoring events cannot provide an accurate picture of stores to Optane NVMM.

While the PMEs seem promising to detect overall usage of NVMM in the case of temporal stores, estimating a consumer's actual bandwidth usage is more difficult due to the lack of further information, such as actual access sizes. We were able to reconstruct the usage of our microbenchmark closely, with only a

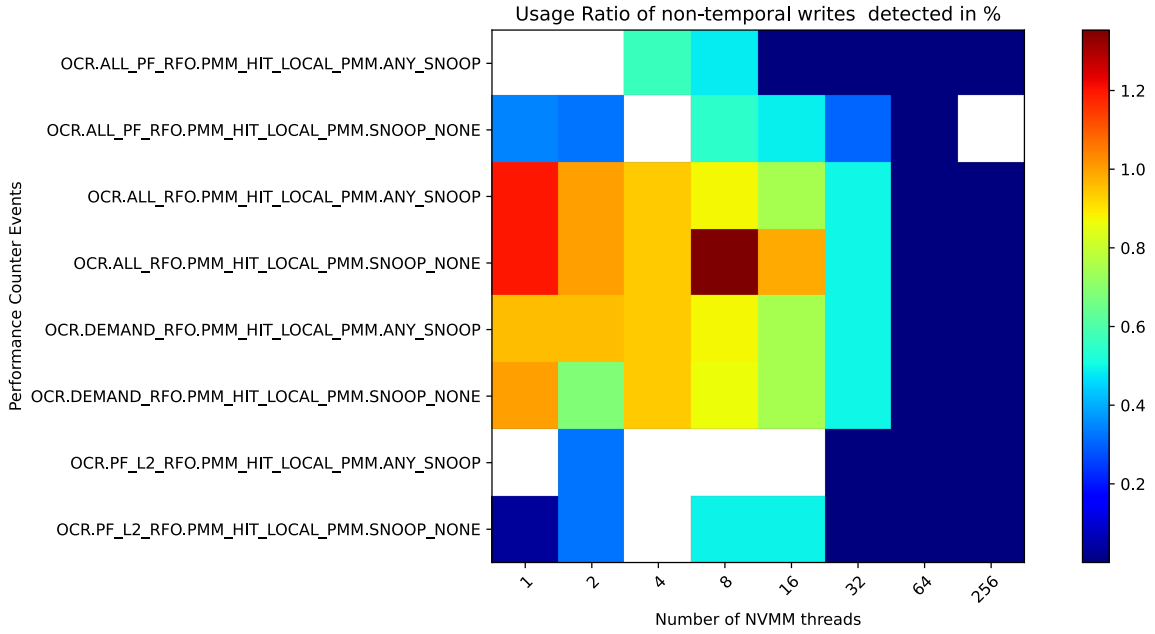

Figure 4.3: Relative detected usage ratio for sequential non-temporal writes per PME at different access sizes. We found the score for the reconstructed usage ratio to be estimatable by the formula mentioned in Chapter 4.3.2. A white field is the result of the PMEs counter returning zero.

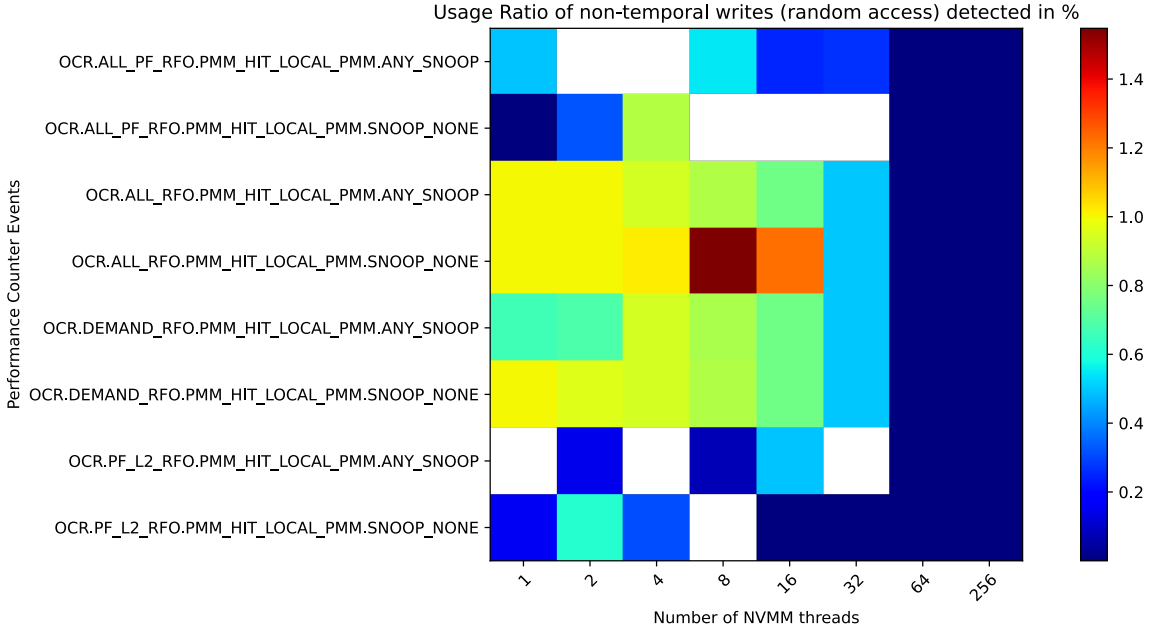

Figure 4.4: Relative detected usage ratio for non-temporal writes with random access pattern per PME at different access sizes. A white field is the result of the PMEs counter returning zero.

few outliers, using a constant and known access size, which leads us to believe that the PMEs are well suited to detect usage of NVMM overall.

Non-temporal stores paint a different picture, as we are critically unable to detect stores of a full cache-line. We assume this is the case, as due to the nature of a non-temporal store, the new contents are not to be placed in the cache after the write. Thus it is sufficient to invalidate the targeted cache line only, not requiring a request for ownership due to partial content changes.

We found the reconstructable bandwidth at a known access size to be increasingly unreliable as the size increases for non-temporal stores.

Applications will generally, to reduce write amplification, attempt to write back blocks as large as possible [35]. They are further likely to use non-temporal stores due to minor cache poisoning and higher bandwidths unless the data is still needed after the write. Both these well-known best practices from Yang et al. [43] cause us to believe that the available PMEs are insufficient.

Thus, we conclude that the selected performance monitoring events would be insufficient to accurately picture the amount of bandwidth consumed per-process in their current form. Even the reliable detection of accesses to NVMM, in general, is questionable considering the importance of non-temporal stores for NVMM.

However, this point may be reevaluated for future hardware revisions that can include other PMEs better suited for our task.

4.5 Future Works

As we have seen, certain performance monitoring events are providing helpful information. In contrast, others are potentially less useful for continuous monitoring due to the limited information inferable and the low number of PMCs available per core. However, it seems reasonable to explore if a combined use of *DEMAND_RFO*- and *ALL_RFO*-based PMEs may be suitable to reduce their divergences from the actual usage.

As future works may also want to use the same approach for reading, possible events should be evaluated. A later step may determine the efficiency in multiplexing load and store related PMEs to use as few of the available hardware PMUs as possible without limiting the overall detection rates for NVMM accesses.

Moreover, as not every process will use NVMM, future works may also consider only configuring the PMU if a to be executed process's context can potentially access an NVMM region.

Last but not least, one may also use the performance metrics of the Optane NVDIMM itself to measure the bandwidth usage at the DIMM and collate this information with the number of writes per process during the same timeframe.

We believe this may help build a heuristic for absolute bandwidth usages in past timeframes.

Chapter 5

PEBS-Based Usage Monitoring

In the previous chapter, we have seen that performance monitoring events for Cascade Lake-based CPUs are neither sufficient to account for writes to Optane NVMM nor to reliably all kinds of stores, underperforming significantly for non-temporal stores. We believe this to be the case as it is a well-known best practice, published by Yang et al. [43], to prefer non-temporal stores and use large stores to reduce write amplification. We expect this scenario may be prevalent in real-world applications.

In their work "HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM" by Rayback et al. [34] the authors describe the use of processor event-based sampling (PEBS) to sample memory accesses that are later analyzed in batches to determine whether a memory page is commonly or infrequently used. Based on those results, HeMem aims to build a multi-tier memory management system that puts "hot" memory pages in DRAM and moves other pages to slower NVMM.

As HeMem shows, PEBS is already successfully used to sample accesses to NVMM in related works. Thus, we decided to utilize it for our following approach to tackle the problem. Each collected sample will contain deeper information about the related memory access, giving us further insights like the precise instruction executed and its destination. This detailed information drives improved insight into the actual workload of a process, likely increasing the accuracy of NVMM accounting. All pre-processed information about the number of accesses and the total bandwidth of each thread and process past NVMM activities can then be stored to drive future policy decisions.

Next, we study the effects of core specialization on the write throughput of Optane DC NVDIMMs. As shown by related works [43], the bandwidth of NVMM is dependent on the number of threads accessing it, at some point decreasing with increasing load. Based on the PEBS-based detection of stores, we affiliate threads with different cores depending on whether they write to NVMM. We expect to see

an increased throughput at a more significant number of threads storing to NVMM compared to an unmanaged environment.

This chapter first discusses the differences between counting and sampling in performance monitoring as relevant for this work. Following is a deeper explanation of the design for our accounting mechanism and an initial attempt to utilize this information for NVMM core specialization. Later, we discuss our implementation of both and continue with the evaluation. Finally, we discuss the results in our conclusion, giving an outlook on future works.

5.1 Differences of Counting and Sampling

Before diving into our design, we first want to highlight some of the key differences between using the processor's performance monitoring units for counting compared to sampling.

Once a configured event occurs, during the counting mode, we increment the counter and can thus make an accurate assessment of the number of occurrences. Yet, we are missing further information about the context of the event. The counting application can later read this counter.

In comparison, we can get further information about each occurrence for a sampled event, which includes the current set of registers, including FLAGS and the instruction pointer. These details are then stored into a *precise event record buffer* allocated and assigned by the application using PEBS. Before the buffer overflows, an interrupt is triggered before the buffer overflows to allow the application time to collect the generated samples. This process overall is expensive; thus, typically, a *sampling rate* is used that is significantly higher than the capture of each occurrence. [11]

Although when using PEBS, we are missing out on a detailed sample for each occurrence, a fixed sampling rate allows us to infer a range of events that happened based on the number of samples we collected. Further, the fixed sampling rate allows us to extrapolate based on the available information, assuming the underlying application's consistent behavior between samples.

In addition, depending on the events in question, the information provided by the samples can be a valuable basis for metrics one may wish to collect that does not have a counting event as an alternative.

5.2 Design

In the following section, we outline an application capable of accounting for writing using a PEBS-based sampling approach. Further, we will continue conceptu-

alizing a lightweight approach to core specialization based on detecting threads storing data on NVMM.

5.2.1 Write Accounting Architecture

Our approach to accounting writes to Optane DC, based on PEBS sampling, starts by configuring one performance monitoring unit per core to sample any executed stores at a configurable sampling rate. By monitoring every available core, we ensure that no thread is scheduled unfavorably and able to elude our monitoring. A thread's sampled stores are going to include accesses to memory that are not NVMM; however, they are filtered out at a later stage based on the details provided.

We determine their executing thread for each of those collected samples, which can be derived from the sample's originating core and its current scheduling information. Further, we gather the registers of a sample containing the instruction pointer and the address of the memory access causing the generation of the sample.

These are passed to a processing thread that infers further information using the originating processes' address space. We use the threads associated process memory map to determine the region the memory access occurred and utilize the region's configuration to infer whether it is mapped to NVMM. If this region corresponds to NVMM, we will continue processing it. Otherwise, the sample is likely caused by access to DRAM and is not relevant for our solution.

Next, we amend the sample by the executed instruction using the instruction pointer. We utilize it and the process' memory map to determine the instructions binary executable and further use a disassembler to learn about the instruction itself and its operands. The instruction's operands let us determine the amount of data stored by it respectively.

This information is then collected in an aggregated form, offering a tuple of the number of accesses and the total detected throughput per thread per second. We further summarize this to gain a per-process view of NVMM usage by associating each thread to its parent process if thread-level would be too fine-granular for the tasks.

5.2.2 Core Specialization

Referencing Gottschlag et al. [6] as related work who successfully performs core specialization for AVX-512, we design a similar solution: We augment the above design with a prototype intending to separate threads triggering stores on NVMM from those that are not.

For this, we categorize a select few cores as *NVMM cores* and restrict threads storing on NVMM to run only on those cores. The separation is achieved by classifying threads as *NVMM-writing* and *not NVMM-writing*.

Threads that are recognized to store to NVMM are temporarily classified into the *NVMM-writing* category and get their thread affinity (see 2.3) adapted to the defined set of NVMM cores only, constraining the scheduler in its placement of these. The classification can expire once a thread is not detected to store on NVMM for a configurable period of time. All other threads can be scheduled to any core available; however, assuming a high load on the NVMM cores, they will be less likely scheduled there by the operating system by default.

5.3 Implementation

We opted for a modular, fully userspace-based implementation, as we deemed it sufficient to evaluate the possible success of our described approach.

This section will be divided into our design section into two parts. First, we will detail our implementation for PEBS-based detection of stores to NVMM and accounting. Then we follow up with our implementation of the core specialization.

5.3.1 Write Accouting Implementation

The part of our implementation responsible for the detection and accounting of NVMM writes consists of multiple modules. The most significant contribution is the *sampler* that acquires the samples and evaluates the necessity to process them further, filtering them out if they are not caused by access to NVMM. Based on those, the *accounting* module updates the thread's current usage after augmenting information, such as the instruction that triggered the sample as well as its access size. Last but not least, the sampler also triggers an update toward the component responsible for our core specialization component *procaffinity*.

Sampling

The sampler is responsible for the fundamental acquisition of samples and determining whether those were to be further evaluated or can be discarded without influencing the accuracy of NVMM accounting.

We start by requesting the number of available CPU cores of the system using *get_nprocs* [18]. Each of the available cores is then configured to provide us with performance information using perf [13].

Initialization Our system call to *perf_event_open()* [20] requests the collection of samples for each of the cores themselves instead of for any preset processes or threads, which is necessary to get a complete picture of all processes running in the operating system. Further, we pinned it to ensure the OS is not going to temporarily remove it to multiplex with other PMEs, improving the detection rates and accuracy of the resulting data. The PMUs are configured with the *MEM_-INST_RETIRED.ALL_STORES* event [9] and we request the perf API to record the instruction pointer, process- and thread-identifier as well as the address of the memory access for each sample. To ensure the instruction pointer is accurate, we further required it to have no skid. Without this setting, the CPU only starts acquiring the instruction pointer after the sample is to be generated, whereas by requesting no-skid, Intel CPUs will capture the IP for soon-to-be overflowing counters. [11] We opted to exclude samples that may occur in a hypervisor or the kernel itself for our perf configuration. This decision was made as we cannot assign actions performed by a transparent hypervisor to any operating system process. Moreover, we decided to exclude accesses by the kernel, as we, as a user-space application, are further unable to detect for which process the kernel and its page cache are currently performing work.

Next, we memory-map a buffer for each core used by perf to place our requested, preprocessed samples. Lastly, a thread is spawned repeatedly, iterating through all core's buffers to collect new samples and process them accordingly.

Sample Processing For every sample collected, we use the provided process identifier to load the process's memory map in question. In addition to that, we look up the processes' memory region at the location of the memory access provided by the sample and determine if the access went to NVMM.

We look at the region's originating path and determine the corresponding mount in the file system's hierarchy. For the moment, we assumed the mount to be to an Optance DC NVDIMM if and only if the the *dax* option (see 2.1.1) is set. Future works may decide to improve on this mechanism.

Should we determine the sample to indeed be for an NVMM access, it is passed on to the accounting module. Nevertheless the result, the sample will be passed on to the core specialization module augmented with the information on whether the access went to NVMM.

Accounting

The accounting module in our implementation is not used to drive any policy decisions for the later-described core specialization, yet only used for verification purposes, printing the detected usage over the last second to stdout. A separate

thread is spawned on initialization that prints the data and then clears the accounting history to output to stdout. However, generally, the accounting mechanism could be used for more advanced purposes with little adaption and is currently only closely built to the requirements of this work.

Determining a proxy IP Once a sample is passed from the *sampler* to the accounting module, we first create a pointer in our tools address space referencing the same data the sample's instruction pointer was pointing to in the originating process. We call this pointer an *proxy instruction pointer*.

To create it, we first reconstruct the sampled processes' address space based on the */proc/{pid}/maps* file [21]. Next, we determined the memory region the sample's instruction pointer referred to and loaded the corresponding static binary file into our memory. After computing the offset of the sample's IP to the start of the memory region in the originating process, we used the calculated offset to reconstruct where exactly our proxy IP had to point based on where our copy of the static binary resides in memory.

Reconstruct the instruction Afterward, we need to reconstruct the instruction itself, which we do use the *capstone* disassembler [33] as a library. At the time of writing, it is recommended to use the long-running feature branch next for purposes of reconstructing our results, as it contains fixes to specific instructions operands metadata one may encounter running temporal stores.

As we previously ensured no-skid on the instruction pointer, we use capstone to reconstruct the instruction by providing it with 20 bytes of data starting at the proxy IP. Should this result in more than one instruction being found, we still only focused on the first one, as it is the causing instruction of the memory access. We know this to be reliable, as we requested the perf API to ensure there is no skid.

As part of the instruction's metadata, we observe its operands. Finally, using the fact that the instruction is from a store to a memory address. We inspect the instruction's operands and determine which writes to memory extracting its size in bytes for the accounting.

Data Structuring Once we determined the size of the write, we update our collected usage information.

These are organized by process identifier, with each entry further broken down into known threads and their identifiers. We increment the number of writes recorded with each sample and added the size to the accumulated total for the current second.

After each second, a thread that was initialized on startup creates a new bucket

for the upcoming second's usage information. The previous second's usage information is then printed afterward and erased from memory.

Optionally, our tool can also be compiled to track the mnemonic of the instruction that caused the memory access. We deemed it necessary to collect the usage details by the process and broken down by thread. Different design patterns, such as applications designed to have separate I/O threads, may otherwise negatively impact their overall performance needlessly.

Caching Both the *sampler* and *accounting* module require introspection into other processes' memory mappings and potentially the contents of their loaded binaries. Thus, we implemented a global memory mapping cache that expires entries if a process has not been seen for the last 60 seconds. This cache consists of multiple layers, first only loading and parsing requested processes */proc/{pid}/maps* file [21], which contains a process's current mapped memory regions and additional information, such as their access rights and origin for the process with the process identifier matching *pid*. The origin, which is often a path in the file system, is later used to determine if a memory access went to NVMM. It can further be used to load and cache executables of the observed process to reconstruct the instruction at the acquired instruction pointers by the accounting module, improving the overall performance.

Moreover, we also keep a cached mapping of all instantiated mounts at */proc/-mounts* and their mountpoints.

5.3.2 Core Specialization

Our *procaffinity* module supported associating threads to groups and enforced these groups to only run on selected CPU cores. It requires the operating process to possess the CAP_SYS_NICE capability. [16]

The *procaffinity* module allowed the creation of groups, represented by integers, that define specific parameters for their member threads. Each group had its own set of processor cores, an expiry time for associated threads, and a stickiness property. Internally, the CPU sets were stored as a *cpu_set_t* data structure [17], which has a significant number of valuable macros surrounding it and can later easily be used to update our threads. On a group's initialization, one defined an expiry time for joining threads, which could be used to automatically unassociated threads that have not been seen for some time. The group *0* was a special default group that threads got associated with when they expire their current, which they could, however, also expire from and then be unassigned from any groups. A possible reason may be that a thread exited. Further, the above-mentioned groupwide stickiness was used to determine if a thread is allowed to be unassigned from a

group before its expiry.

To associate a thread to a group, the operational process invokes *procaffinity* and passes the thread identifier and its expected group association. This association is expected to happen frequently to reconfirm that a thread is still associated with its current group, refreshing the expiry. It is checked if the thread was already assigned to any other group on the association. Should that be the case, it would leave the old group and join the now expected one, setting the threads' expiry to the default of the newly assigned group. However, this was only possible if the last group was not marked as sticky, in which case the thread association would not be updated. On the other hand, if the thread is already part of the group, the expiry is updated as if the thread had just joined. Further, if ultimately the thread joins or switches groups, its thread affinity is updated by calling *sched_setaffinity* [23], passing the *CPU_SET* associated with the new group.

For simplicity, the expiry is checked for every thread associated with any group on any update or assignment attempt. This was a reasonable decision, as they happened very frequently. We used *procaffinity* to update a threads' group association for each sample we found, regardless if it was an access to NVMM.

Configuration For our use of core specialization, we used *procaffinity* with two groups. The first group, which was also the default group, assigned each process a *general CPU set* and kept processes assigned for up to 60 seconds. The benefit of this group is that it allowed us to be on the lookout for threads scheduled on ony core due to our *sampler*, then associate them all with our potentially more restrictive set of CPU cores. For instance, we used this mechanism to ensure the load of the entire system could be concentrated on cores from one NUMA node to provide our evaluation will not be impacted. If a thread exited or just has not been seen for 60 seconds, it defaulted to any core available to the system again.

Our second group was assigned to any thread that performed an NVMM store. The group was sticky and had an expiry of 200ms. These cores were a subset of the general CPU set of *group 0*. Once a thread was not detected to use NVMM up to the expiry, it was released to use a more extensive set of cores again.

5.4 Evaluation

In this section, we first introduce you to our testing methodology in our evaluation of the PEBS-based usage monitoring and core specialization tool. Next, we will discuss how accurate our solution is in terms of reconstructable throughput and its consistency. Finally, we will focus on the core specialization and how our minor implementation can affect the runtime of adjacent software and its impact on the observed NVMM bandwidth.

5.4.1 Testing Methodology

We want to ensure NUMA effects will not influence our results, similar to our evaluation of the performance counter-based approach. Thus, we deliberately restricted our benchmark to cores sharing the same NUMA node as the single, non-interleaved Optane DC DIMM we used. All benchmarks below effectively had 16 CPU cores available.

Further, they were run with the Best Practices by Yang et al. [43] in mind. We configured the *pmm-writer* to use non-temporal stores at 256-byte access size to achieve the highest possible bandwidth, which was typically at about 1.9 GB/s. While these kinds of stores were indetectable for the counter-based approach, we will see them be easily so using the PEBS-based method.

Significant variables during the below benchmarks are the *sampling period*, which describes how often the event had to occur for us to get one sample where we opted to test at *2503*, *5107* and *9973*. Moreover, the number of *writer threads* and, in particular, for core specialization, the *number of cores in our NVMM group* out of 16 cores in total we have available for one NUMA node are of higher importance.

Again, similar to the evaluation of the performance counters, all benchmarks were run five times, and their results were averaged to have a reliable data basis. Further, we note that all the below benchmarks were run using the same user-space application designed to associate the NVMM usage to processes while also managing a threads association to the group of nvmm-utilizing and other applications based on our core specialization approach.

5.4.2 Store Detection

We start by determining how well our sampler is able to capture the stores from processes writing to NVMM. A high detection rate is preferred, as with it, a better estimation of the actual NVMM usage per process is possible.

The benchmark starts our PEBS-based usage monitor, listening for activity on all cores. Then, we run from 1 to 16 *pmm-writer* in parallel, each configured to write precisely 32 GB onto NVMM. Once all those writers finish, we stop the usage monitoring.

In terms of accuracy, we considered two domains to be of significant value to dive deeper into. First, we determine how well our implementation can reconstruct the throughput of the writers. Then, we further discuss the consistency of those results.

Reconstucting Usage

Similar to Section 4.3.2, we tried to determine how much of the written data we were able to reconstruct in terms of usage. As the PEBS-based approach allowed us to know the access size s for each individual access, we know the observed usage $U_o = \sum is_i$ for each thread. We previously defined a formula for the ratio of reconstructed usage u_r, which we are now able to simplify to:

$$ReconstructedUsageRatiou_r = \frac{ObservedUsageU_o}{TotalUsageU_t}$$

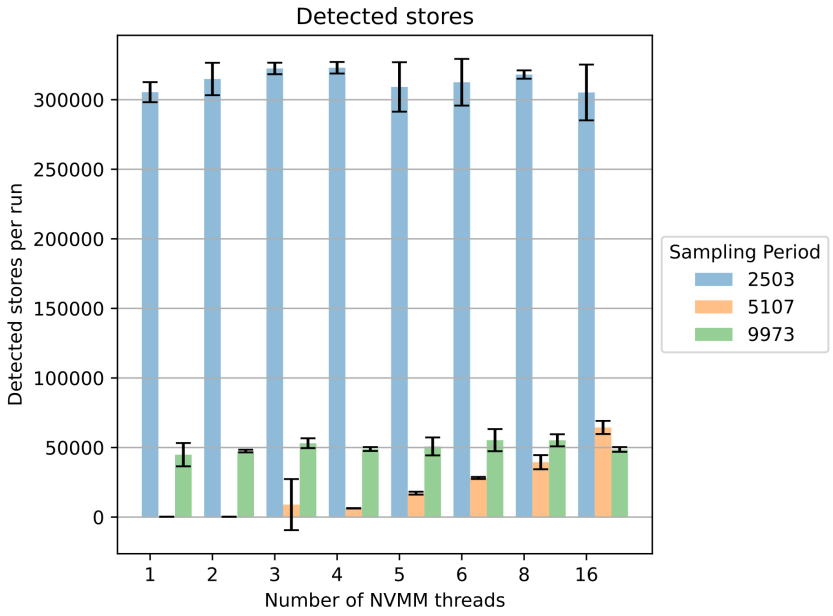

Figure 5.1: We can see the amount of accesses detected at different numbers of parallel NVMM writer threads for different sample periods normalized by the number of threads. The sample period has a significant effect.

Figure shows the averaged number of stores detected at different amounts of NVMM writer threads operating in parallel divided by said thread count to normalize the graph. Further, we can see how the different sampling periods influenced these results.

As is apparent, the detection rates show slight variation for different numbers of NVMM threads, except the sampling period *5107*, which starts off near zero and is increasing as the writer thread count grows. The average detected samples at 2503 were about *304596* whereas we detected *44758* at a rate of 9973. This

informs us that there is no bottleneck due to the higher number of threads, as the detected writes would otherwise decrease for increasing thread counts.

While focused on the number of stores detected, we also verified the reconstruction of the instructions to work reliably. We used AVX-512 vector extensions for our stores of 256-bytes; these were effectively four stores each of 64-bytes. For all our run benchmarks, the total throughput observed was an exact multiple by 64 of the number of stores.

Interpolating by sample period By interpolating the detected stores by the sample periods, we estimate the absolute number of stores. However, this value has to be seen with caution. First and foremost, by interpolating, we assume the thread to be performing the same task for an extended period, which may not reflect real-world scenarios. As we know, each thread wrote exactly 32 GB and did not perform other activities; we can proceed in the case of our benchmark.

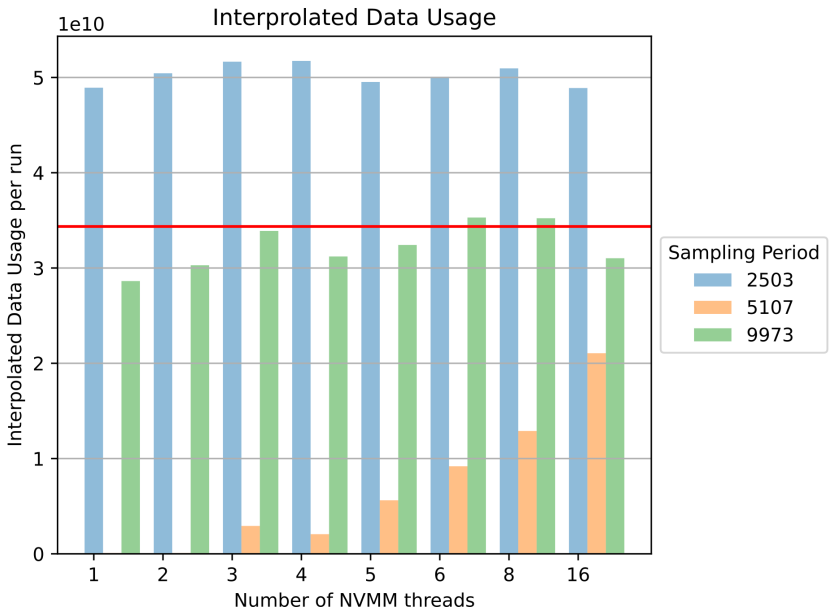

Figure 5.2: The number of accesses found in Figure 5.4.2 was multipled by the sampling rate. We show this information for different thread numbers and sampling rates. The results show statistically to be expected interpolated data usage. The red line represents to target of 32 GB.

Figure 5.2 shows us the interpolated data usage we are able to infer statistically as the number of detections from Figure 5.4.2 multiplying by their sampling rate. The visualized red line shows our actual data usage of each thread at 32 GB.

We find that at the sampling rate of 2503 we consistently overestimated the data usage, on average overestimating by 42%. In the case of the sampling period being 9973, we on average underestimated by 16.85% but generally were closer to the expected result. Finally, as one expects the interpolated at the sampling rate of 5107 is still unstable.

Consistency

We wanted to understand if our sampling was consistent and equally detected NVMM stores of all threads. This is a fundamental requirement for meaningful results, as a more significant deviation from the mean could indicate an unfair preference of certain threads during the sample acquisition.

We inspected individual runs and found no such significant deviation in any benchmark performed.

5.4.3 Core Specialization

As the first application for our PEBS-based store detection, we implemented a core specialization for NVMM writing threads.

To determine how this approach performs, we constructed and defined an *unmanaged* scenario, in which the benchmark and NVMM writer threads are run solely managed by the kernel's scheduler. Further, we define a *managed* scenario in which, before the benchmarks start, our PEBS usage monitor is started, which integrated our core specialization solution. Thus, threads that are detected to store to NVMM will be assigned to an NVMM group with only a subset of CPU cores available to them.

Should we be benchmarking a managed scenario, we first started with the PEBS usage monitor to run in the background. Then, regardless of the scenario, we first start a set of *pmm-writer* processes followed by a CPU benchmarking software *sysbench*, [36], running a multi-threaded prime benchmark with 16 threads up to the number 10000000. Once the benchmark is completed, we stop all *pmm-writers* and the monitor, if it is was running.

In this section, we will first evaluate the runtime impact of our interference with the default scheduling of the operating system. Next, we will look at the resulting changes to the total bandwidth towards the Optane NVDIMM.

Impact on Benchmark Runtime

We set out to see how CPU-bound processes are affected by an increasing number of NVMM writing threads. To quantify the overhead, we summed the total running time of all our unmanaged *sysbench*'s threads.

Figure 5.3 plots the summed runtimes of *sysbench*'s threads, comparing the unmanaged to the managed scenario. Before the point of NVMM writing threads saturating the number of CPU cores available is reached, the runtime of *sysbench* is slowly but steadily rising, ranging from 5% to 30% for every doubling. Once 16 NVMM storing threads were active, saturating the number of available CPU cores, we find the unmanaged scenario's total runtime to increase by about 50% for every doubling of NVMM writer threads.

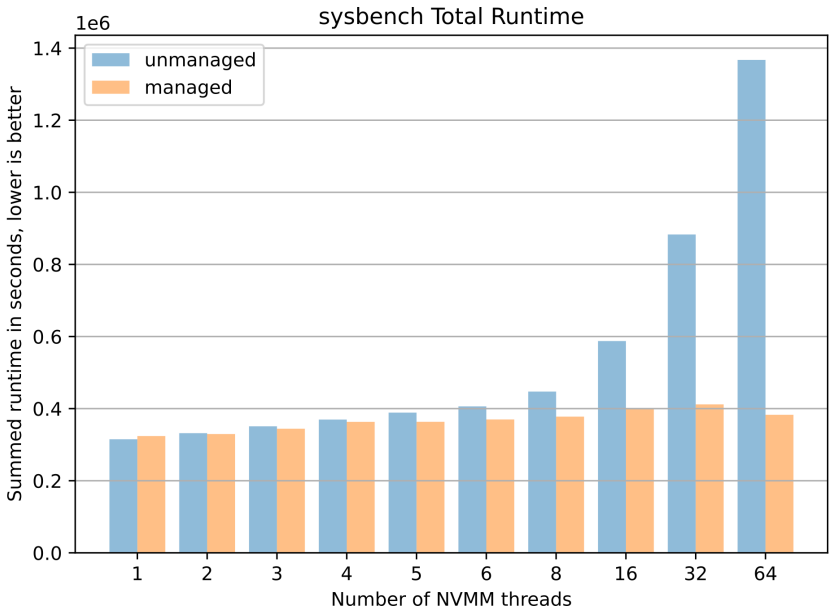

Figure 5.3: We compare the total duration of the *sysbench* [36] benchmark's threads runtimes aggregated for an unmanaged workload toward a managed on. The graph shows the results at different numbers of NVMM writer threads. We find our core specialization to keep the runtime of the benchmark consistent.

In contrast, the summed runtimes are barely rising in the managed scenario. The total increase compared the quickest with only one NVMM writer with sysbench competing against 32 NVMM writers is just 27,12%. At the point of writer saturation, we find the managed solution to allow sysbench to perform better by 31,01%.

To conclude this, we find it unsurprising that the runtime is only slowly increasing for the managed scenario. As all NVMM writing threads are consolidated by the core specialization on three cores. The remaining are freely available to sysbench. Likewise, a non-neglectable increase in runtimes is expected for the unmanaged case due to the increasing competition on shared CPU cores.

Impact on Optane Throughput

Parallel accesses to Optane DC are known to result in diminishing bandwidth after peaking at up to four threads. [43] Constraining a multitude of NVMM threads to a limited number of cores could impact the achievable bandwidth by reducing the degree of parallelism, thus effectively increasing the achievable bandwidth.

Figure 5.4 shows the average throughput observed during the benchmarks for unmanaged and managed scenarios in comparison.

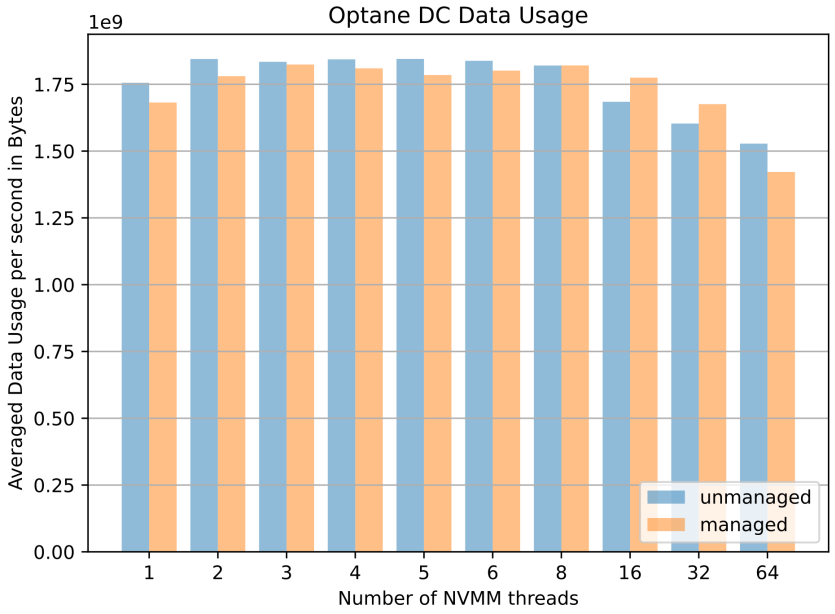

Figure 5.4: We show the aggreagted throughput per second of all NVMM writer threads whilst the *sysbench* [36] benchmark was running for an unmanaged to a managed workload.

While we can observe the typical decrease in bandwidth in our unmanaged scenario, the peak can be observed at five threads, which is slightly later than previous works found [43]. This may be due to the high system load due to the CPU-bound sysbench running parallel. Further, we find our first more considerable drop in bandwidth at the point of NVMM threads saturating the available CPU cores, which continues decreasing at about 5% for every doubling of the NVMM threads afterward in our observation.

As for managed, the peak is found at three threads, showing a neglectingly higher bandwidth than we saw as the peak for unmanaged.

Directly comparing unmanaged and managed, we observed bandwidths are close with the highest discrepancy at one thread by 4.2%; however, for both cases,

the bandwidth has still not peaked yet. Aside from this, unmanaged is up to 3.4% faster up to 8 NVMM threads, from which point managed takes the lead up to 32 NVMM threads by up to 5.36%. We see this as a confirmation that decreasing the degree of parallelism using core specialization is a potentially valuable tool.

Sample Processing Overhead

Finally, we take a closer look at the overhead caused by the sampler, including the accounting, as we see that retrieving the access size is vital information for every sample. While as long as the system has enough computing resources available, one could argue that this procedure's impact is negligible, it is no longer the case under load, as during our benchmarks.

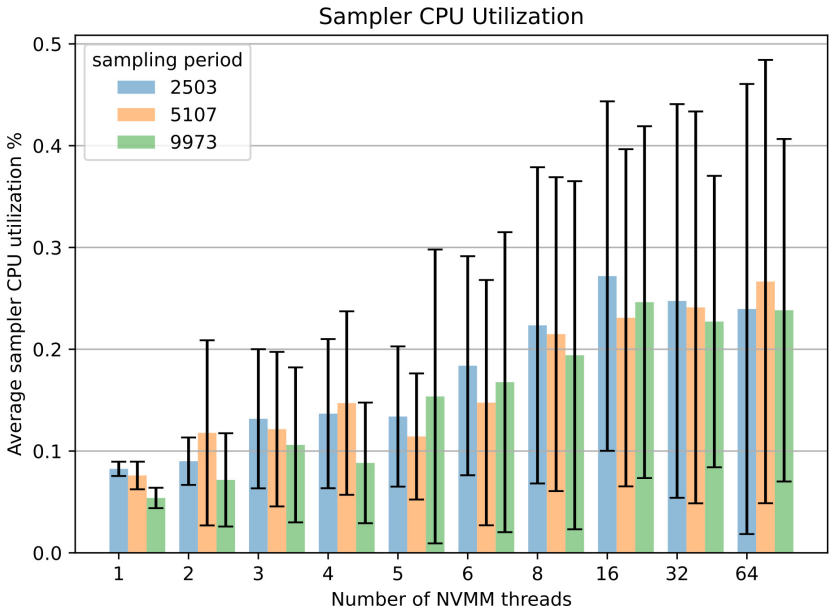

Figure 5.5: We show the overhead caused by the sampler and accounting module during our benchmark at variying NVMM thread counts and sample rates. The sampling rate has little effect compared to the NVMM threads.

In Figure 5.5 we see the overhead of the procedure by the total wall-clock time of the benchmark. The graph breaks this information down by NVMM thread count and sampling size. We can observe the thread count has a noticeable impact on the overhead, which is expected as the overall workload increases.

As for the sampling period, we again discover it to have a more minor impact than one may anticipate. While we can see a minor reduction with increasing

sample periods for the same thread counts, the impact is nowhere near a halving of the incurred overhead.

Last but not least, we notice a significant standard deviation in our underlying data basis for these measurements, increasing with rising NVMM thread counts.

5.5 Conclusion

We have seen our implemented NVMM usage monitor work reasonably well, providing consistent results detecting stores to Optane DC NVMM. It became however apparent that the selection of the sampling period is a key point and should be further investigated.

As for the grouping of NVMM cores onto a subset of available cores, we have also seen good results. Adopting NVMM usage for process scheduling can reduce the impact on adjacent CPU-bound processes by up to 31% for a matching number of NVMM threads and available CPU cores.

Moreover, the negative impact on the total bandwidth of NVMM is small, at most 4.2%, and could be viewed as evened out by an in comparison higher bandwidth at larger thread counts of up to 5%. The observed shift may cause core specialization to be the start of a software-based solution to improve the possible parallelism of NVDIMMs in the future.

Finally, regarding our implementation, we believe a kernel-based implementation may be more rewarding, in particular, to address shortcomings we are unable to solve from the user space, such as the detection of read and write system calls to NVMM-based storage. We stand by this decision for now as NVMM-based applications will likely shift to mapping-based persistent storage compared to the continued use of system calls for the mentioned performance improvements possible with direct access in Section 2.1.1.

5.5.1 Limitations and Future Works

Concerning our implementation, there are a few minor things future works may improve on. First, we assume a mount point to be NVMM only if *dax* is provided as a mounting option. While this will likely be fulfilled, it is not a necessity and could be implemented more reliable by checking the device path and correlating with *ndctl list* [38]. Further, we retrieve mounts from */proc/mounts*, however, Linux introduced per-process mounts some time ago, which could result in our implementation being unable to detect certain accesses to NVMM. [21] Also mount details are not currently updated during the runtime of our NVMM usage monitor, which could also be addressed by future works.

A more significant limitation is missing support for just-in-time compiled code, which should be introduced by future works to support a wider range of real-world applications.

Moreover, future works may port our work to the kernel space which could improve the accuracy and overhead caused by our approach. As already discussed, a kernel-based implementation could be able to detect reads and writes to NVMM performed via system calls, and precisely account for those. Further, similar to our recommendation with the counter-based approach, we would like to see the collection of samples stopped if a thread is scheduled that has no NVMM mapped into its address space.

It would also be valuable to have a similar PEBS-based setup for the detection of reads from NVMM and how the observation of both directions could be combined.

Finally, as for our evaluation, we would like to see future works take a closer look at the use of different sampling rates, possibly evaluating with real-world applications.

Chapter 6

Discussion

We contributed a usage monitor with reliable detection of stores to NVMM. The PEBS-based usage monitor provides us with samples we can use to reconstruct the executed instruction to infer its access size. Based on this, we can statistically interpolate the NVMM usage of each process. Related works, such as *NThread* [40], would benefit from this as it allows to estimate a process's usage even for memory-mapped NVMM regions.

The integrated core specialization assigns NVMM threads to a subset of the available CPU cores, resulting in a reduction of parallel threads writing to the NVDIMMs. We observed an increase in the overall throughput at higher thread counts due to this adjustment and see operating systems potentially utilizing such mechanisms in the future. We believe it eases the design of NVMM-utilizing applications by reducing the need to limit stores to only a few threads, as works like *SPMFS* [44] needed to adopt to achieve high bandwidth.

As for future generations of Optane DC, we would like to see Intel work on supporting a higher number of threads accessing NVMM in parallel. One possible short-term solution may be to increase the XPBuffer, as this would reduce the contention observed by related works [43].

However, in the longer term, we would like to see performance monitoring events that are specifically designed to introspect the utilization of NVMM by tasks running on the CPU. Ideally, we would like a PME that provides the total read/write utilization since it has been configured. Alternatively, it would be valuable to have PMEs available in counting mode to determine how many accesses occurred. We would prefer the events to be further configurable to observe different access sizes individually to reconstruct the throughput.

As for sampling-based usage estimation, instead of relying on a PME that tracks all accesses to memory, we would like to see one in particular for accesses to NVMM. We believe this would significantly reduce the overhead of our current implementation while increasing the accuracy of throughput estimations.

Chapter 7

Conclusion

We have established that available Optane DC's performance is highly impacted by the access pattern, as small random accesses cause significant write amplification. This write amplification combined with limited XPBuffer size reinforces a low degree of parallelism each NVDIMM is capable of without reducing the overall performance. Solving this issue requires the operating system to become NVMM usage aware, which is difficult as applications prefer using direct access to reduce the overhead incurred.

We evaluate the use of performance counters to detect an application's access to NVMM, both in counting and sampling configurations. After finding a set of viable performance counter events that can be used to count stores to NVMM, we evaluated these further. We found counting works out reasonably well for temporal stores, it is however difficult to reconstruct non-temporal stores, becoming almost impossible for non-temporal stores of a size of 64-bytes and multiples. Further, we found it likely to be difficult to reconstruct the size of the occurred accesses in a practical application.

In contrast, our contributed sampling-based approach built on top of PEBS is able to detect these stores reliably. We use it to acquire samples for stores to memory, then filter out all stores that do not target memory residing on NVMM. Using the instruction pointer collected with each sample, we are able to reconstruct the executed instruction and determine an accurate size for each access, which in turn was used to build a framework for accounting per-process NVMM usage.

Moreover, we used the sampler to implement a user-space prototype of a core specialization, grouping threads storing onto NVMM. Those were then only provided a subset of the available CPU cores for their execution. We found this management to result in a significant improvement in throughput for other compute-intensive tasks residing on the system. Further, reducing the set of utilizable CPU cores for threads writing to NVMM resulted in an improved throughput to NVMM at higher thread counts than in an unmanaged case.

Finally, we discussed how related works could already benefit from implementations similar to ours and steps Intel could take in the future to help improve throughput for highly parallel workloads and estimations for per-process accounting of NVMM usage.

Bibliography

[1] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. *SIGPLAN Not.*, 51(10):677â€“694, oct 2016. https://doi.org/10.1145/3022671.2984019.

[2] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGARCH Comput. Archit. News*, 39(1):105â€“118, mar 2011. https://doi.org/10.1145/1961295.1950380.

[3] Intel Corporation. Intel optane dc persistent memory. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf, accessed on 2021-07-18.

[4] Intel Corporation. Technology brief | intel optane technology: Memory or storage? both. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/what-is-optane-technology-brief.pdf, accessed on 2021-07-18.

[5] Mingkai Dong, Qianqian Yu, Xiaozhou Zhou, Yang Hong, Haibo Chen, and Binyu Zang. Rethinking benchmarking for nvm-based file systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2967360.2967379.

[6] Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. Automatic core specialization for avx-512 applications. In *Proceedings of the 13th ACM International Systems and Storage Conference*, May 2020.

[7] Red Hat Inc. Chapter 5 using nvdimm persistent memory storage red hat enterprise. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_storage_devices/using-nvdimm-persistent-memory-storage_managing-storage-devices, accessed on 2022-03-21.

[8] Intel. intel/intel-pmwatch. https://github.com/intel/intel-pmwatch, accessed on 2022-04-29.

[9] Intel. perfmon-events 2nd generation intelÂ® xeonÂ® processor scalable family based on cascade lake product. https://perfmon-events.intel.com/cascadelake_server.html, accessed on 2022-04-30.

[10] Intel. Intel xeon processor scalable memory family uncore performance monitoring, jul 2017.

[11] Intel. Intel 64 and ia-32 architectures software developerâ€™s manual, combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4, dec 2021.

[12] The kernel development community. Direct access for files. https://www.kernel.org/doc/html/latest/filesystems/dax.html, accessed on 2021-08-29.

[13] kernel.org wiki. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, accessed on 2022-04-29.

[14] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. Pactree: A high performance persistent range index using pac guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 424â€“439, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3477132.3483589.

[15] Linux man-pages project. sched_setaffinity. https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html, accessed on 2022-03-21.

[16] Linux Programmer's Manual. capabilities(7) - linux manual page. https://man7.org/linux/man-pages/man7/capabilities.7.html, accessed on 2022-05-10.

[17] Linux Programmer's Manual. Cpu_set(3) - linux manual page. https://man7.org/linux/man-pages/man3/CPU_SET.3.html, accessed on 2022-05-10.

[18] Linux Programmer's Manual. get_nprocs_conf(3) - linux manual page. https://man7.org/linux/man-pages/man3/get_nprocs_conf.3.html, accessed on 2022-05-08.

[19] Linux Programmer's Manual. mmap(2) - linux manual page. https://man7.org/linux/man-pages/man2/mmap.2.html, accessed on 2022-04-28.

[20] Linux Programmer's Manual. perf_event_open(2) - linux manual page. https://man7.org/linux/man-pages/man2/perf_event_open.2.html, accessed on 2022-05-08.

[21] Linux Programmer's Manual. proc(5) - linux manual page. https://man7.org/linux/man-pages/man5/proc.5.html, accessed on 2022-05-08.

[22] Linux Programmer's Manual. read(2) - linux manual page. https://man7.org/linux/man-pages/man2/read.2.html, accessed on 2022-04-28.

[23] Linux Programmer's Manual. sched_setaffinity(2) - linux manual page. https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html, accessed on 2022-05-10.

[24] Linux Programmer's Manual. write(2) - linux manual page. https://man7.org/linux/man-pages/man2/write.2.html, accessed on 2022-04-28.

[25] micron. Bridging the gap between dram speed and nand nonvolatility, 2013. http://www.micron.com/products/dram-modules/nvdimm, accessed via web.archive.org.

[26] Microsoft. Understand direct access (dax) and create dax volumes with persistent memory devices, jul 2021. https://docs.microsoft.com/en-us/windows-server/storage/storage-spaces/persistent-memory-direct-access, accessed on 2022-03-22.

[27] Andrzej Nowak and Georgios Bitzes. The overhead of profiling using PMU hardware counters, July 2014. https://doi.org/10.5281/zenodo.10800.

[28] Deok-Jae Oh, Yaebin Moon, Eojin Lee, Tae Jun Ham, Yongjun Park, Jae W. Lee, and Jung Ho Ahn. Maphea: A lightweight memory hierarchy-aware profile-guided heap allocation framework. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2021, page 24â€“36, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3461648.3463844.

[29] Intel OpenSource. /perfmon/clx/. https://download.01.org/perfmon/CLX/, accessed on 2022-04-30.

[30] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 304â€“315, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3357526.3357568.

[31] Andy Rudoff Piotr Balcer. pmem/pmdk - github. https://github.com/cpmem/pmdk, accessed on 2022-05-09.

[32] Intel PR. Intel and micron produce breakthrough memory technology, jul 2015. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/, accessed on 2021-08-05.

[33] Nguyen Anh Quynh. capstone-engine/capstone - github. https://github.com/capstone-engine/capstone/tree/next, accessed on 2022-05-08.

[34] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392â€“407, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3477132.3483550.

[35] Christian Schwarz. Low-latency synchronous io for openzfs using persistent memory, June07 2021.

[36] Monty Taylor. sysbench. https://launchpad.net/sysbench, accessed on 2022-05-10.

[37] PMDK team at Intel Corporation. Ipmctl user guide. https://docs.pmem.io/ipmctl-user-guide/, accessed on 2021-08-25.

[38] PMDK team at Intel Corporation. Ndctl introduction. https://docs.pmem.io/persistent-memory/getting-started-guide/what-is-ndctl, accessed on 2021-08-25.

[39] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. *SIGOPS Oper. Syst. Rev.*, 25(5):26â€“40, sep 1991. https://doi.org/10.1145/121133.121140.

[40] Ying Wang, Dejun Jiang, and Jin Xiong. Numa-aware thread migration for high performance nvmm file systems. In *36th International Conference on Massive Storage Systems and Technology (MSST 2020)*, 2020.

[41] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508, 2020.

[42] Vincent M. Weaver. System-wide performance counter measurements: Offcore, uncore, and northbridge performance events in modern processors, jul 2017. https://web.eece.maine.edu/~vweaver/projects/perf_events/uncore/offcore_uncore.pdf, accessed on 2022-03-28.

[43] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/fast20/presentation/yang.

[44] Yang Yang, Qiang Cao, Jie Yao, Yuanyuan Dong, and Weikang Kong. Spmfs: A scalable persistent memory file system on optane persistent memory. In *50th International Conference on Parallel Processing*, ICPP 2021, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3472456.3472503.

[45] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: A key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 194â€“209, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3447786.3456237.