저작자표시-비영리-변경금지 2.0 대한민국

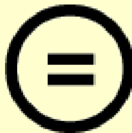
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

M.S. THESIS

NF-log: Revisiting log writes in relational database for efficient persistent memory utilization

비휘발성 메모리 환경에서의 관계형 데이터베이스 로그 시스템 연구

August 2022

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

ZGUEM SARA

M.S. THESIS

NF-log: Revisiting log writes in relational database for efficient persistent memory utilization

비휘발성 메모리 환경에서의 관계형 데이터베이스 로그 시스템 연구

August 2022

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

ZGUEM SARA

NF-log: Revisiting log writes in relational database for efficient persistent memory utilization

비휘발성 메모리 환경에서의 관계형 데이터베이스 로그 시스템 연구

지도교수 염헌영

이 논문을 공학석사 학위논문으로 제출함

2022 년 4 월

서울대학교 대학원

컴퓨터 공학부

ZGUEM SARA

ZGUEM SARA의 공학석사 학위논문을 인준함

2022 년 6 월

위 원 장	장 병 탁	(인)
부위원장	염 헌 영	(인)
위 원	전 병 곤	(인)

Abstract

Non-volatile memory (NVM) is a promising storage technology that combines not only high performance and byte-addressability (like DRAM) but also durability (like SSD). However, as existing relational database management systems (RDBMS) are originally designed based on the assumption that all the data and log are stored on high latency block based devices, they are not able to take full advantage of this new technology yet. Consequently, write operations will under-utilize the device(NVM) and eventually downgrade the performance. In this work, after properly analyzing the redo-log mechanism in InnoDB and tested its impact on the overall system performance. We propose NF-log, a redesigned log critical path that aims to eliminate redundant flushes to the disk by eliminating the impact of page cache and adapting the appropriate APIs. After that, we eliminated the remote persistent memory overhead and optimized the write ahead mechanism to reduce memory copy overhead by adjusting the WA-mechanism triggering threshold and granularity. Our design utilizes the NVM byte addressability and its persistence feature to reduce the write size by 30% and boost up the performance to up to 38% for sysbench write_intensive workloads and up to 16% for TPC-C.

Keywords: Storage, Database, Non-volatile Memory, Relational Database, Log optimization

Student Number: 2020-28751

Contents

Abstract i

Contents ii

List of Figures iv

Chapter 1 Introduction 1

1.1 Motivation . . . 3

1.2 Contribution . . . 6

1.3 Outline . . . 7

1.3.1 Related Work . . . 8

1.3.2 background . . . 10

Chapter 2 Implementation and Design 17

2.1 Analysis of InnoDB Log System . . . 17

2.1.1 Concurrent Writes . . . 17

2.1.2 Checkpoint . . . 20

2.2 Design Goals . . . 21

2.3 Immediate persists . . . 22

2.4 Improve data locality . . . 25

2.5 Improve write ahead mechanism 26

Chapter 3 Evaluation **30**

3.1 Experimental Setup 30

3.2 Performance breakdown 31

3.2.1 Evaluating immediate persist optimization 32

3.2.2 Evaluating write ahead optimization 33

3.2.3 Evaluating data access locality optimization 34

3.3 Overall Performance 35

3.4 Resource utilization 39

3.5 Write Size 43

Chapter 4 Conclusion **45**

요약 **51**

List of Figures

Figure 1.1 Effect of redo log on the overall system 5
Figure 1.2 Memory hierarchy 10
Figure 1.3 Redo Log Physical Structure 12
Figure 1.4 Write performance comparison with and without remote persistent memory access 15

Figure 2.1 Redo log thread concurrency 18
Figure 2.2 Redo log critical path 23
Figure 2.3 Redo log thread concurrency(updated) 25
Figure 2.4 Write ahead mechanism in InnoDB 26

Figure 3.1 Evaluating immediate persist optimization with sysbench write only workload 32
Figure 3.2 Evaluating write ahead optimization with sysbench write only workload 33
Figure 3.3 Evaluating data access locality optimization with sysbench write only workload 34
Figure 3.4 Sysbench overall performance 35
Figure 3.5 Sysbench Read Intensive Performance 37

Figure 3.6 Peformance with TPC-C Bechmarks, 100W, NEWORD(2.8%), PAYMENT(2.8%), OSTAT(2.8%), DELIVERY(45%), SLEV(45%) 38
Figure 3.7 Peformance with TPC-C Bechmarks, 100W, NEWORD(5.5%), PAYMENT(5.5%), OSTAT(0%), DELIVERY(89%), SLEV(0%) 39
Figure 3.8 Peformance with TPC-C Bechmarks, 750W, NEWORD(5.5%), PAYMENT(5.5%), OSTAT(0%), DELIVERY(89%), SLEV(0%) 40
Figure 3.9 InnoDB 1million write transactions PMEM utilization . 40
Figure 3.10 NF-log 1million write transactions PMEM utilization . . 41
Figure 3.11 InnoDB 1million write transactions DRAM utilization . 42
Figure 3.12 NF-log 1million write transactions DRAM utilization . . 42
Figure 3.13 InnoDB 1million write transactions DRAM utilization . 44

Chapter 1

Introduction

Non-Volatile Memory (NVM) is becoming a popular storage device for research due to its DRAM-like byte-addressability and disk-like durability features. Even though NVM technologies such as PCM, STT-RAM and ReRAM have been researched for several years, they remained as experimental products and their features are slightly different with each other [1]. However, Intel has released its commercial level product recently: optane DC persistent memory, which performance is verified and stable. It is expected to perform 2-3.7x slower and fulfill 1/3 of DRAM bandwidth for loads, but 1/6 bandwidth of DRAM and match the latency regarding to stores [2].

Although the novel properties of NVM are highly relevant to Relational databases, they also present new challenges. Existing RDBMS is either designed based on traditional disk-based architecture or modern in-memory architecture, none of them can fully utilize the features of NVM devices yet. Among all the challenges, optimizing the logging mechanism is crucial. Being a necessary component to all RDBMSs, redo log preserves data atomicity and durability.

However, existing log mechanisms of RDBMS are designed on the assumption of storing data on block devices. With the features of NVM, such as supporting byte-addressbility with low latency differ from the page-addressability and higher latency when dealing with block devices, using existing log mechanism on NVM will introduce unexpected overheads. To solve the problem, M.A.Ogleari et al proposed [3], which is a hardware undo+redo logging scheme that maintains data persistence by leveraging the write-back, write-allocate policies. In addition, Steven Pelley et al [4], introduced a group commit mechanism to persist transactions' updates in batches to reduce the number of write barriers on NVM. Other optimisations [5–9] not only eliminate lock contentions due to a centralized log buffer but also include check-pointing techniques and in-cache-line undo logging.

In this paper, we mainly demonstrate how database management systems can benefit from log mechanism optimizations when operating on optane DC memory by making modifications on a MySQL storage engine(InnoDB). The reason we choose InnoDB [10] is mainly for its wide usage range and transactional properties that other MySQL engines do not support. We evaluated InnoDB with Intel optane DC memory and found there is a huge performance degradation compared to the promised optane DIMM performance due to the non-optimized access, therefore the under-utilization of the features of the device. We conducted detailed analysis to the internals of InnoDB and we observed certain lacking that needed to be addressed in its logging component.

We evaluated our work by using TPC-C and sysbench. the result shows in the case of TPC-C, the performance in terms of transactions per minute, has been improved by up to 16% and by 39% when running sysbench after making optimizations, our resource utilization experiments demonstrate NF-log's cost effectiveness and it's ability to reduce the total write size.

The main issues of the existing InnoDB redo log are: ①the overhead of page cache on the overall system.②the coarse-grained write ahead buffer alignment and ③ the overhead of remote persistent memory access.

The rest of this paper is organized as follows: In section 2, we will talk about existing approaches to integrating optane DIMM to RDBMS. Then we will demonstrate our analysis results and design details in section 3 and present evaluation results in section 4. Related work will be described in section 5 and finally we conclude this work in section 6.

1.1 Motivation

To demonstrate the problems addressed in this paper we will start by some context to fully utilize the byte addressability and memory mapped IO it is recommended to mount the device in DAX mode. When accessing PMEM in this environment it is best to utilize memory mapped IO which is much more straightforward than traditional IO through bypassing the page cache.

As described in section , In step 2 once the log buffer is full the log_writer thread is triggered and the log buffer contents are written to the page cache. In PMEM environment, when the log_writer wants to write the data to the page cache, it will use the read()/write() APIs, it will call some other Kernel APIs that try to access the page cache (and since it is not available) it will access the device directly. After that the log flusher will call the sync again to flush the same data. Meanwhile, the session threads remain in standby for these 2 operations to be performed before we can write more data, resulting in redundant flush overhead. Intel has proposed the PMDK library which allows us to utilize the option of memory mapped I/O, with the use of these libraries, we were able to redesign the critical redo log path and alleviate that system

overhead.

We have performed an experimental micro-benchmark study to understand the characteristics of optane DC memory. In this experiment we compared the performance difference of Optane DC memory with different IO modes. One is when we map the file in the Optane DIMM to the address space directly and flush to it using PMDK interface [11] varying the access granularity, and the other is when we let the operating system handle the mapping and address translations and flush the data to the persistent memory using the traditional pwrite() interface. From one to thirty-two threads can be dispatched to each modify it's own file(1GB of data). We can choose the write size among five options to be dispatched every time(64B, 512B, 1K, 4K, 32K) and (16M, 2M, 1M, 250K, 31K, 25K) write counts respectively for each thread.

After running several rounds of this benchmark we have made some performance related observations. We noticed that generally, with smaller sequential write size, PMDK outperforms pwrite() at both cache line and page granularity and the optane DC memory will saturate at around 8 threads. For larger writes though, file IO will catch up and they will all start yielding similar results. Highlighting 512byte write size, since it is redo block size in InnoDB, and especially when deploying one thread(log_flusher), we notice that adapting PMDK libraries yields better throughput than the traditional file IO interface for both random and sequential write formations. With larger write sizes, even though the performance of both interfaces is very similar, traditional file IO still has the upper hand in throughput performance.

After that we test how much the log write performance will affect the overall performance of InnoDB. To verify that, we ran an experiment where we placed both the redo log and data files in the optane DC memory and then moved the redo log files to a slower device and compared the performance. We got the

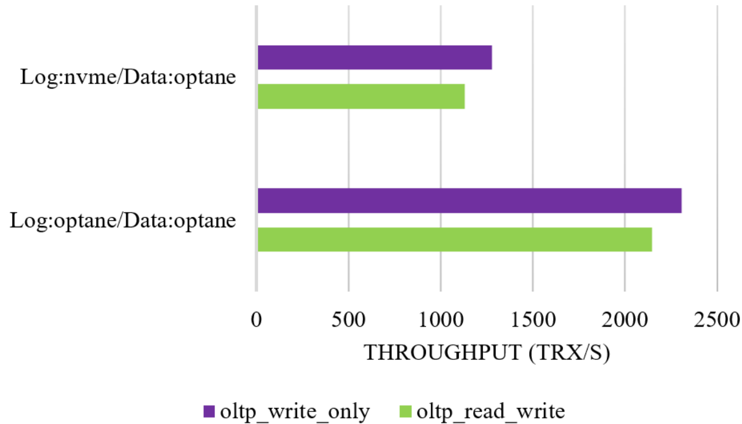


Figure 1.1 Effect of redo log on the overall system

results represented in figure 1.1 from deploying 32 threads, which is our best performance, and we noticed that we lose approximately 60% of the write_only and read_write performance just from moving the log file to a slower device.

Other than that, the choice to bypass the page cache in our memory architecture proves that the original architecture of conducting separate write and flush operations to improve InnoDB performance on block devices is unnecessary as it incurs redundant writes to and persists from the system page cache. The same situation for the write ahead buffer, it was included in the original structure as a bridge between the log buffer and the page cache to align log writes and avoid additional reads from the disk to the system page cache. However, optane DC memory has a different internal access granularity compared to block devices. On top of that, since system page cache has not been a part of our memory hierarchy anymore, read-on-write issues will occur in the internals of optane DC memory rather than in the system page cache. Therefore, we saw the need to improve the latter motioned feature. Finally, after seeing the performance degradation resulted from remote persistent memory access, we saw

the need to optimize data locality.

Following the redo log investigations and the baseline results mentioned above, we decided that the current InnoDB is not optimized to take advantage of the features of optane DC memory, matched with its overall importance on the whole system performance, we saw the need to revisit the design and implement a redo mechanism for more efficient persistent memory utilization.

1.2 Contribution

The main contributions of this work start with the detailed analysis of the log system within InnoDB. As a complicated system that is fairly optimized to block devices, our paper is one of few resources when it comes to the analysis and optimizations done on the internals of InnoDB. Since we are using optane DIMM with DAX(Direct Access) mode, and InnoDB being built on the assumption of operating on block devices, comes the need to adapt its logging system to better utilize the features and environment of optane DC memory.

Therefore we proposed NF-Log(Non-Flushing log), an optimized logging component in InnoDB with no redundant flushing, improved data access locality and efficient write ahead mechanism. Currently, InnoDB uses system page cache to bridge the access latency gap between log buffer and block devices. However, Since optane DIMM has DRAM-like latency, using system page cache will introduce overhead into the system instead. The problem mainly lays in the redundant persists performed from the page cache to the NVM after data has been written to the device directly from the log buffer, bringing significant overhead over the whole system. To alleviate the impact of page cache on our new environment, we invalidated the writes to the system page cache combined with the separate persists to flush the data to the device. Instead, we redirected

the writes to persist to the optane DIMM immediately after being written to the log buffer. This Optimization will remedy to the redundant log data persists to the optane incurred by separating the write and the flush in the original implementation, therefore improve the logging performance.

Also, further investigations led us to the fact that current write-ahead mechanism is using an 8KB write-ahead buffer to avoid read-on-write issues that usually happen on page-cache miss for block device. However, this is also introducing some overhead when running on optane DIMM as it is byte-addressable in comparison to the page addressability of the block device. Therefore, we re-designed this write ahead mechanism not only by adapting the alignment granularity to optane DIMM's byte-addressbility but also eliminating the extra data copies to the WA buffer. In addition, we observed that remote persistent memory access in NUMA architecture will cause severe performance degradation. We also implemented some changes to improve data access locality within InnoDB. We evaluated our work by using TPC-C and sysbench. the result shows in the case of TPC-C, the performance in terms of transactions per minute, has been improved by up to 16% and by 39% when running sysbench after making optimizations.

1.3 Outline

The rest of this paper is organized as follows:

- **Chapter 2** we will talk about related work and existing approaches to integrating optane DIMM to RDBMS.
- **Chapter 3** we will demonstrate our analysis results and design details
- **Chapter 4** present evaluation results

- **Chapter 5** summarizes and concludes our work. It also points out the directions for future work.

1.3.1 Related Work

In this section we discuss researches related to log-write optimizations and other RDBMS optimizations to operate on persistent class memory.

Regarding to log related optimizations, ELEDA [12] proposed a solution that can help move a surge of log data safely from the volatile memory without risking atomicity not durability. It enables using multicore parallelism to operate a highly concurrent data structure that helps eliminate synchronous IO delays. Meanwhile, [3] presents an undo+redo logging scheme which maintains data persistence by leveraging the write-back, write-allocate policies used in commodity caches. This solution manages to get its improvement not only from relaxing the ordering constraints con caches and memory controllers, but also by implementing hardware logging and forcing write-back. In addition, Write Behind Logging (WBL) [13] proposed a mechanism that flush the data to the database before being recorded to the log, that way enables a DBMS to recover from system failures by knowing which parts of the database have changed rather than how it was changed. There are also other logging optimisations not only eliminate lock contentions due to a centralized log buffer to alleviate overhead [5], but also include check-pointing techniques and in-cache-line undo logging [6].

On the other hand, regarding to RDMBS optimizations on non volatile memory, [14] provides that disk oriented DBMS's do not particularly perform worse than in-memory RDBMSs when using non volatile memory as both systems continue to assume that memory is volatile. [15] provides an extensive analysis of relational database engine behavior with Intel optane DC memory.

This analysis focuses on configurations that magnify hardware difference by increasing the amount of IO done within the critical path and the effect of knobs on performance. They also provided details on the behavior of different types of queries and operators on PCM. In addition, [16] compared the performance between persistent memory and DRAM, they analysed the potentiality of using PCM as volatile memory for data indexes in memory optimized tables (MOT) engine. But they failed to extend this research to exploit failure atomicity. And [17] proposed special data structures that can fit PCM's feature.

In the topic of developing specifically NVM aware DBMS logging mechanisms. [4] introduced a group commit mechanism to persist transactions' updates in batches to reduce the number of write barriers required for ensuring correct ordering on NVM. SOFORT [18] proposed a hybrid SCM-DRAM storage engine that speeds up restarts by taking advantage of the properties of SCM to operate on the persisted data directly without having to first cache it in DRAM. It also speeds up recovery by bypassing traditional log and updating the persisted data in place in small increments. Besides, ReDu [19] brings hardware assisted logging mechanism and a design that further exploits a small region of DRAM as a write-cache to remove NVM writes from the critical path.

The design of NF-Log based on the features of Intel Optane DIMM such as its internal access granularity and parallelism that analyzed in [14,15]. The key idea of NF-Log that persist log data immediately aligns with the mechanism that proposed in [4,13], which is reduce the write critical path and the latency of writing data. In addition, our NF-Log also take the data locality into consideration to prevent remote NVM access that can decrease the performance. Finally, NF-Log proposed the idea that using small write ahead threshold to reduce the hardware level read-on-write problem.

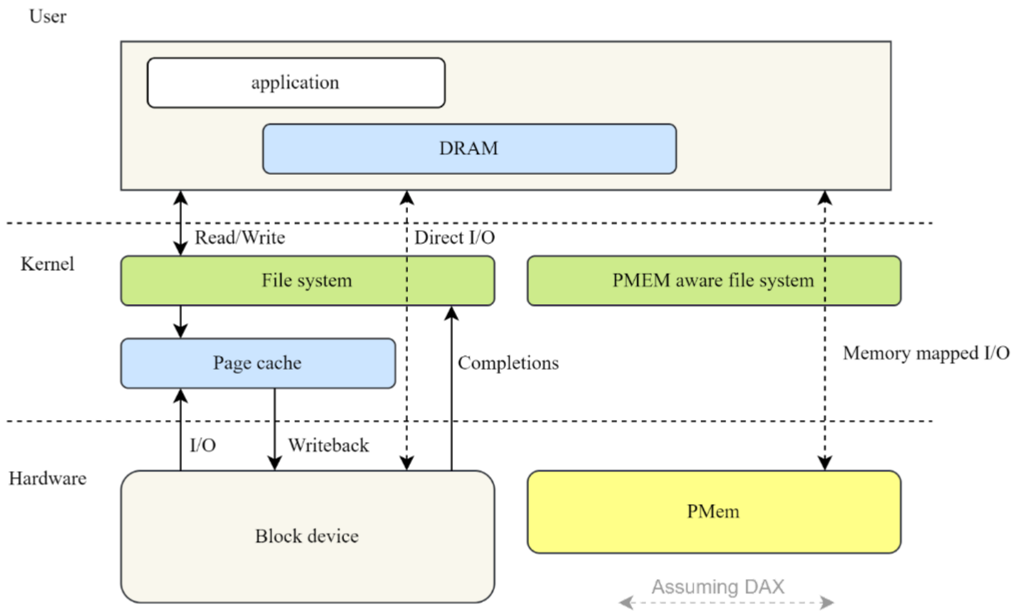


Figure 1.2 Memory hierarchy

1.3.2 background

We will begin by introducing the key features of optane DC Memory in section 1.3.2 including its price and performance comparison with DRAM. Then we will give an overview analysis of the redo log within InnoDB.

Intel optane DC Memory

After the commercialization of the first non volatile memory device by Intel, the optane DC persistent memory is now almost 10 times cheaper than DRAM. To be more specific, 512GB of optane DC memory is 43% percent cheaper than 256GB of DRAM and 256GB of optane DIMM is priced 44% lower than the 128GB DRAM.

Performance wise, optane DIMM will evidently have slightly higher latency and lower throughput compared to DRAM. For reads, DRAM generally peaks

at 39.4 GB/s, and for writes, it takes just four threads to reach saturation at 13.9 GB/s. For a single optane DC memory, its max read bandwidth is 6.6 GB/s, whereas its max write bandwidth is 2.3 GB/s. [20].

Optane DC Memory has two configuration modes [21]: ① In-memory mode, where the DRAM + optane DIMM will represent a large volatile main memory unit. The DRAM will act as a cache to hold the most frequently used data and the optane DIMM will help supply larger capacity of the main memory. The memory controller will perform all the predictions of the data with the higher cache-hit rate and promote them to the faster memory. As this is helpful in exploiting the large capacity of persistent memory for cheaper price. However, it is also significantly slower than operating solely on DRAM.

The other option that used in our work is the ② App Direct mode shown in figure 1.2 where the CPU can perform two characteristically different load and store operations to both the DRAM and the Optane DIMM separately. Operations that do not need to be persisted and require lower latency will be serviced in the DRAM and structures that need to be persisted will be transferred to the optane DIMM. When using this mode, applications can directly access the device bypassing the page cache by mounting a file system in DAX mode, and by using PMDK (Persistent Memory Development Kit) [11] we can directly map the files in the device to the virtual space of the related applications.

Stores to optane DIMM are pulled from the asynchronous DRAM refresh and sent to the optane DIMM in 64B cache-line size granularity. After address translation the actual access to storage media occurs. As the optane physical media access granularity is 256B, the optane controller translates smaller requests into larger 256B accesses, causing write amplification where small stores become read-modify-write operations.

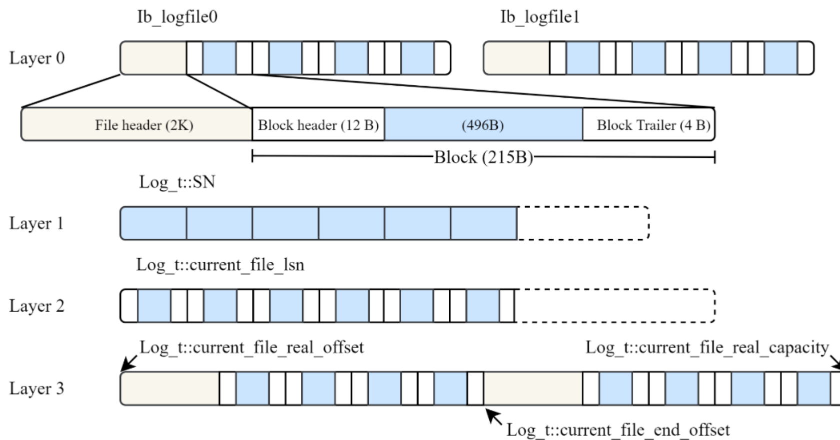


Figure 1.3 Redo Log Physical Structure

InnoDB Log Writes

Mini Transactions(MTR)

InnoDB will generally split the transaction execution process into several MTRs [22]. Each MTR will handle a series of operations such as locking, writing redo log, writing data and releasing locks. During the life time of a mini transaction, a log of changes is collected inside an internal buffer of the MTR. It contains multiple log records, which describe changes applied to possibly different modified pages. When the MTR is committed, all the log records are written to the log buffer within a single group of the log records.

Redo Log Overview

In this paragraph we will introduce some brief features of redo log. InnoDB will write the log sequentially and the redo log of each MTR is appended to the end of a fixed-size file. Therefore, we will have to switch to the next file when the current file is full. The number of redo log files is also fixed and it will have a

2KB FileHeader per file. After the last space of the last file has been written, it will wrap around to the first file to continue writing. After the FileHeader, and as represented in layer 0 of figure 1.3. The file is divided into blocks of 512B, each block contains a 12B BlockHeader, a 4B BlockTrailer, and the contents of the actual redo log in the middle.

Conceptually, there is a globally incremented SN (sequence number) and LSN(log sequence number). SN corresponds to the serial number of all written redo log original content, and LSN is the serial number after the original content including both the BlockHeader and BlockTrailer. These two can be converted to each other.

Log file offset translation

InnoDB will generally write the redo log data into more than one file, each not only containing solely the original log content, but also the metadata related to it. In this paragraph we will explain how the data is organized into the files and how we can browse through said file contents through LSN conversion. As demonstrated in figure 1.3, there are two redo log files in the example, ib_logfile0 and ib_logfile1, each of which is 4KB (including 2KB FileHeader and four 512B Blocks).

In the first layer, there are consecutive blocks making up the original content of the redo log to be written, and SN is the sequence number of the original content. InnoDB will divide the original content into units of 496B, add a 12B BlockHeader and a 4B BlockTrailer to each unit, forming a 512B redo-log Block. This structure is similar to the second layer in figure 1.3. It will use current_file_lsn to index the sequence of the previously mentioned blocks.

The current_file_lsn is the logical growth of the actual content of the redo log, so theoretically it can increase continuously to UINT64_MAX without looping

back. However, since there are limited number of redo log files, current_file_lsn needs to be aware of file change and wrap back to the beginning after all files are written. In this case, the conversion between the layers shown in figure 1.3 is required. Since there are only 2 files in our example, each of which is 4K, the upper limit of the actual content that can be written is 8K. After removing the FileHeader (4K) of the two files, the actual effective content is 4K, which indicates that current_file_lsn will switch to the next file every time it increases by 2K, and wraps around to the first file every time it increases by 4K.

One thing to note here is that redo log component is not aware of a single file. It does not explicitly switch file descriptor when switching files. The redo log considers the following two files logically continuous as represented in the third layer of figure 1.3. We use current_file_real_offset to represent the offset within log file, with an initial value equal to 2048 (the front is the FileHeader of the first file). Initially current_file_real_offset and current_file_lsn correspond, that is 2048 $\leftrightarrow$ 8192, each subsequent write updates these two values synchronously, and the mapping conversion from the logically infinite current_file_lsn to the actual finite current_file_real_offset can be completed. In addition, current_file_end_offset represents the end position corresponding to the file currently being written. If current_file_real_offset exceeds this position, a 2KB Header is added to it to switch to the next file.

Representing the sum of the actual sizes of the two files is files_real_capacity. If current_file_real_offset exceeds current_file_real_capacity value, it means that the current two files have been written, and it needs to rewind to the first file to rewrite. Here, the current_file_real_offset will be reset to 2048 to complete the rewind.

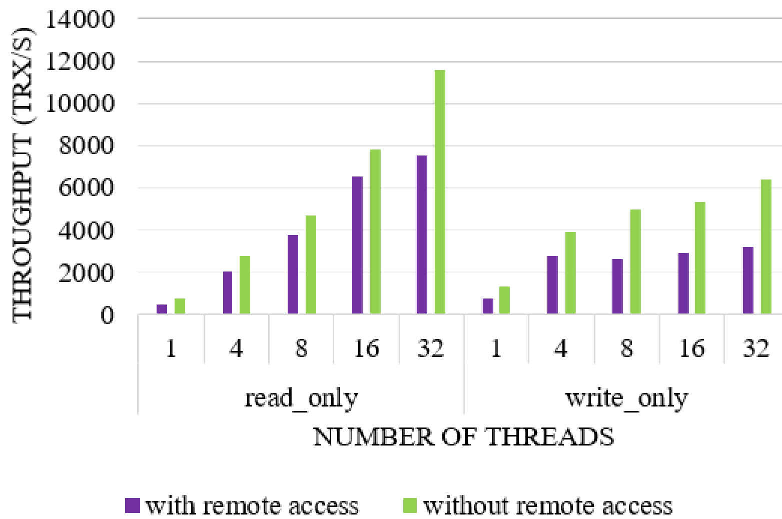


Figure 1.4 Write performance comparison with and without remote persistent memory access

Write Log

The redo log will pass by a few steps before it reaches the storage device. To be able to manage these writes and make them concurrent we will have to use different types of threads to handle different steps of the log persisting process.

①Session threads will first write the contents to the log buffer and notify the log_writer thread to continue.

② One log_writer thread will transfer the data from the log buffer to the page cache using pwrite IO interface and return to to notify the waiting session threads.

③After that one log_flusher thread will be notified and triggered to flush (fsync) the contents of the page cache to the storage device. After finishing, the log_flusher will notify the session threads waiting for its completion.

Data locality

Meanwhile, NUMA architecture has been widely used due to it's scalability. However, remote memory access latency overhead is also its well known disadvantage. Accessing memory not belonging to the current CPU socket will cost us more latency than accessing local memory. Since optane is also using DIMM interface, how its performance will be affected by remote persistent memory access is intriguing. To verify its performance variation, we intentionally turn off one of the CPU sockets to simulate the environment of non-remote optane DIMM access in our test server and running sysbench on the top of InnoDB. Figure 1.4 demonstrates the results of our experiment, from the results we can see that remote access will slow the system by 53% for reads and by 104% for the writes.

Chapter 2

Implementation and Design

2.1 Analysis of InnoDB Log System

Before making the adequate optimizations on the system, we needed to establish the current design and architecture of the redo log component, following to the information mentioned in Section 1.3.2 we will present a detailed explanation of the state of art of InnoDB redo log component:

2.1.1 Concurrent Writes

Redo log writing in InnoDB is an append write. In theory, when multiple MTRs append their own redo log at the same time, some synchronization mechanism is required to ensure write order. In our used InnoDB version here, a lock free mechanism is used. In that version, each MTR can get the actual log file offset that it needs to write on before writing the actual log, and reserve that space, therefore there is no need for using lock to strictly guarantee the order of the redo log write. To explain more in detail what was mentioned in Section1.3.2,

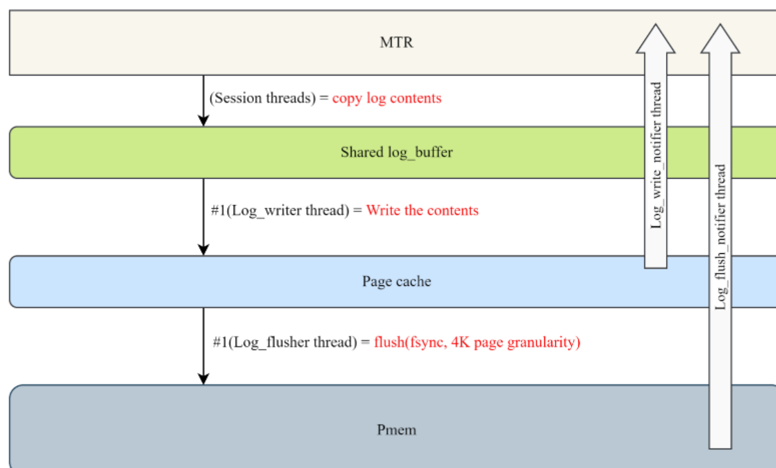


Figure 2.1 Redo log thread concurrency

every log write is usually written at minimum size of 512B and typically passes by 3 steps:

① First, there is a fixed size log buffer in memory. each MTR will first update Log_t::sn to get the sn (sequence number) range corresponding to its own log, and then passes SN→lsn→log buffer position conversion, and copy the log contents to the log buffer. This is performed by session threads and notifies a log_writer thread to take care of the next step.

② The log_writer thread is responsible for continuously scanning the log buffer and searching the new continuous content that has been written since the last write position. Then, it will write this continuous content to the page cache. In the original design, the log_writer is the one constantly detecting and writing the log buffer contents to the page cache, therefore making it the thread that actually writes to the file.

③ The flushing of the redo log is also done asynchronously. The log_flusher thread will detect whether that new data is being written to the page cache

but has not been flushed yet, then it will complete to flush this part of the log to the storage device.

We have five types of threads participating in the redo log process:

- log_writer
- log_flusher
- log_write_notifier
- log_flush_notifier
- log_closer

There is also multiple condition variables involved in synchronizing between these threads:

- writer_event
- write_events[]
- write_notifier_event
- flusher_event
- flush_events[]
- flush_notifier_event

In this section, we will describe the working mechanism of most two important types of threads, which are log_writer and log_flusher, their behavior is demonstrated in figure 2.2. At the beginning, the session threads that execute MTRs write their logs into the log buffer concurrently. If there is no remaining space in the log buffer before writing, it will wake up the log_writer thread

waiting on the writer_event to write the log buffer data into the page cache and release log buffer space. During this period, the session threads will wait on write_events[], until the log_writer thread wakes them up after writing to the page cache. After the session threads have been woken up, which means that the current log buffer has more space to write the redo log corresponding to each MTR. The session threads will copy it to the corresponding position of the log buffer, and then update it on recent_written corresponding interval mark. After this, the corresponding dirty pages are hang on the flush list, and the corresponding interval mark on recent_closed is updated. This will be needed for checkpoint later.

log_writer mechanism: The log_writer will wait for the session thread to wake up on the writer_event. It will scan recent_written after waking up to check whether there are new continuous logs in the log buffer after write_lsn, and then write them into the page cache together. Finally, it wakes up the waiting threads, which are either the session threads waiting on write_events[] or log_write_notifier thread waiting on write_notifier_event. Eventually, it will wake up the log_flusher thread waiting on flusher_event.

log_flusher mechanism: It will wait for the log_writer thread or other session threads to wake it up on flusher_event. At first, it compares the flushed_to_disk_lsn of the last flush with the write_lsn currently written to the page cache, if it is less than the latter, it will incrementally flush the contents to the disk, and then wake up possible session threads waiting on either certain flush_events[] or waiting for log_flush_notifier on flush_notifier_event.

2.1.2 Checkpoint

When InnoDB determines to execute checkpoint, it will check three different variables: First is the recent_closed.m_tail, which means that the dirty pages

corresponding to the previous lsn have been hung on the flush_list. Second is that it will take the lsn with the smallest oldest_modification on the flush_list, which means that the dirty pages corresponding to the previous lsn have been flushed to the disk. And finally, flushed_to_disk_lsn means the redo log corresponding to lsn has been flushed to the disk before this. It will start by comparing the first and second value, and use the minimum value out of the two. It is then compared with flushed_to_disk_lsn to get the minimum value. If this value is greater than the current checkpoint_lsn, this value can be used as the new checkpoint_lsn. In short, the checkpoint is the minimum lsn for which both the data and the corresponding redo log have been placed on the disk.

2.2 Design Goals

According to the previous analysis, we can see how meticulously InnoDB was designed to utilize several aspects of the running environment from the block device features to the memory hierarchy, carefully harmonizing its path to alleviate system overhead. However, by choosing to use Intel optane DC Memory, we are bringing significant changes on such environment. Our database is currently being processed fully in-memory. When we say in-memory here, we are not directly referring to the main memory only but also Intel optane DIMM that represents the lowest level of our memory hierarchy bypassing the page cache.

To alleviate overhead mentioned above, the design goals of NF-Log are:

1.Remove redundant persists. According to the original InnoDB, when executing an DDL/DML, it firstly writes data to the page cache at then persists all the data at once to increase bandwidth utilization. In this case, page cache is used to bridge the latency gap between the DRAM and the block device. However, with optane DC Memory, and with the help of immediate persist functions

provided by PMDK, the considerations implemented on the page cache are not only unnecessary, but are also creating a significant overhead by performing a second persist during the data flush phase. It is important to eliminate this impact on the overall system by flushing data immediately at an appropriate granularity.

2. Improve data locality. NUMA architecture is widely used to improve system scalability. However, the disadvantage of accessing remote memory on NUMA architectures is also well acknowledged, and since optane has an even larger access latency than DRAM, we see the need to ensure data locality by preventing remote optane memory access.

3. Improve the current write ahead mechanism. InnoDB's Write-ahead mechanism aims to reduce read-on-write issues, thus improving I/O throughput. However, not only it incurs an unnecessary copy to the page cache(not part of our memory architecture), but we also think that the current granularity of write ahead mechanism, which is 2x page size (8KB), is coarse-grained and unfit for optane DIMM environments. As a result, we need to reduce the size of the WA alignment to better utilize the finer-grained addressability of the device and get closer to it's internal transfer granularity.

2.3 Immediate persists

As was mentioned in Section 2.1.1, every log write takes multiple steps and passes by 3 main write and flush operations: in first step, a write is from the MTR's buffer to the log buffer, and in step 2, from the log buffer to the page cache and then it is finally flushed to the disk in step 3. But as we use Intel optane DC Memory mounted with DAX mode, system page cache is bypassed. Therefore we need to also make modifications on the critical path of the log, invalidating the page cache considerations and performing a persist to

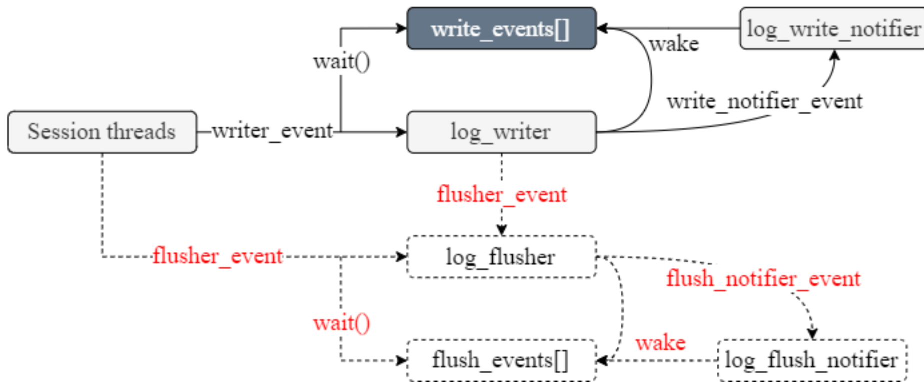


Figure 2.2 Redo log critical path

the disk immediately from the log buffer. This operation will be handled by the log_writer thread. Not to mention the overhead that occurs to the session threads that remain in wait status for the log_flusher to finish performing the unnecessary persist to the storage device.

We utilized PMDK libraries and interfaces [nontemporal] which allowed us to alter the design of the thread concurrency model(figure 2.2) in InnoDB. NF-log implementation will consist of two steps instead of three, which are: writing the log content to the log buffer by the session_threads, and then flushing immediately to the optane DIMM. This entails excluding the log_flusher and the log_flush_notifier from the redo log critical path. We have disabled what is shown as a dotted link in figure 2.2. Before going about that, we made sure to implement the data persists procedure into the log_writer critical path.

As we already mentioned, to improve performance of concurrent execution, all the modifications inside InnoDB are established by using MTRs. To ensure the atomicity, MTR uses redo logs and will commit if and only if the redo log contents are written into disk. Originally, pwrite() is used to write the log data into the page cache and then the log_flusher thread will call (fsync) to persist the data from the page cache to the hardware cache of the optane and that's

when it will be able to return to the waiting session threads and wake them up. By using PMDK, we are able to perform the copy and the persist immediately in the log_writer context. In addition to that we synchronize the events and disable the session threads wait on the log_flusher to continue. And then end with updating all the related lsns. After verifying the correct functioning of our system we went ahead and disabled the unnecessary threads.

PMDK provides several libraries and interfaces to use in order to handle data management in the persistent memory. We have compared the yielded throughput by three data moving instructions:memmove(), memcpy() and pmemset() and two different persist operations: non_temporal and persist. After running some evaluations on all the possible combinations, we noticed that they all perform very similarly, and the combination of memmove() and non_temporal gave us slightly better throughput, so we chose to use it in our implementation.

To summarize this section, previously, when we use the read()/write() APIs to write data from the log buffer, it will call some other Kernel APIs that try to access the page cache, and since it is not available -bypassed at DAX mode- it will access the device directly. Later on, we will call another flush to perform the same operation all over again while session threads are waiting for completion. In figure 2.3 we can observe the updated redo log path after eliminating redundant persists, smaller granularity and non_temporal flushes that bypass the CPU cache. In our implementation we have used Persistent Memory Development Kit API's which allowed us to utilize the option of memory mapped I/O and alleviate that system overhead. This path is not only optimal for log writes, but also does not affect the reads at all. We remind that we are dealing with Redo log which represents stale data that doesn't need to be cached and is only read at recovery. Another detail, is that when we bypass other caches such as CPU cache. the System will have to perform longer latency access instead

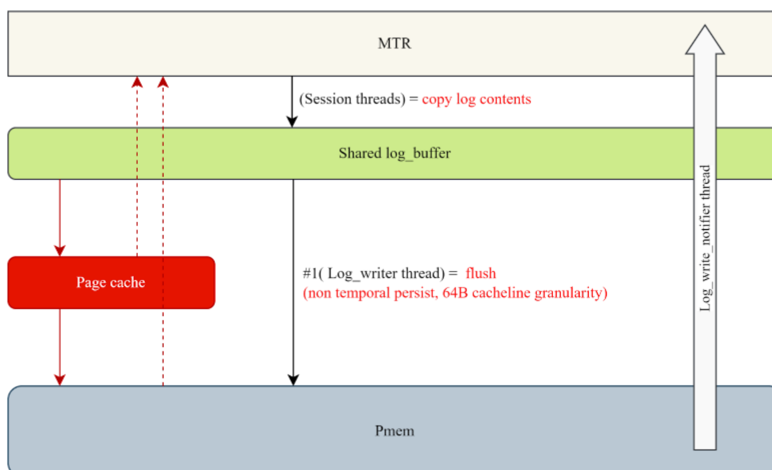


Figure 2.3 Redo log thread concurrency(updated)

of caching the data, but we argue that as data writes can be "written back" asynchronously. The redo log on the other hand has to be persisted immediately all the way to the storage device in orderly manner, which coincides with the access rules of NVM.

2.4 Improve data locality

Currently, most of the servers are using NUMA architecture due to its scalability. But as we mentioned in section 1.3.2, accessing optane DIMM from remote CPU cores will cost at most 104% of write performance degradation in InnoDB. Therefore, avoiding remote optane DIMM access is also necessary to improve the performance of InnoDB.

In our design, we use optane DIMM as a storage device, which means all the data that will be processed within InnoDB is persisted in the optane DIMM, including redo log, undo log, double write buffer files and data files. Each type of file is processed by a group of threads, for example, as we mentioned in section 1.3.2, that there are five types of threads involved in the redo log processing

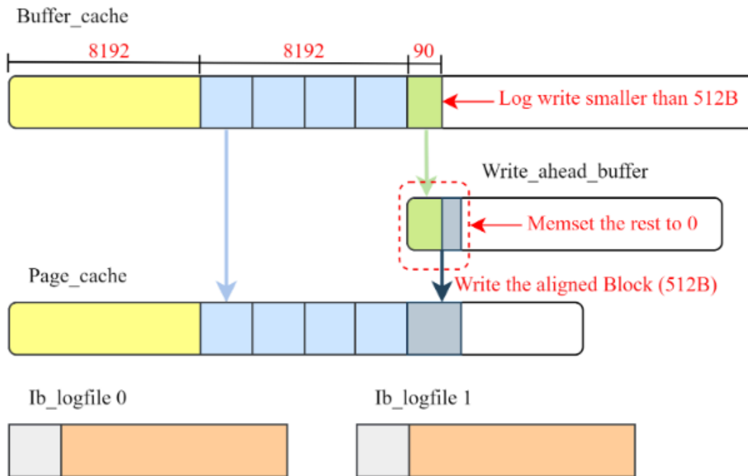


Figure 2.4 Write ahead mechanism in InnoDB

and three of them are directly accessing the log data.

Our idea to solve data locality issues is to statically pin the data accessing threads to the same CPU socket where the optane DIMM that stores our accessed data is installed. Note that in NF-Log, we currently pin the threads to the whole socket and not to a certain core as that will severely limit the CPU utilization. In the future, we plan to enhance this design by dynamically pinning I/O threads to different cores in the same CPU socket where the optane DIMM containing our data is plugged according to the current CPU utilization(Improved load balancing).

2.5 Improve write ahead mechanism

As we mentioned before, current InnoDB is designed based on the assumption of storing data in a traditional block device. Since the access latency between general block devices and DRAM is large, page cache is used to bridge the latency gap between them. The operating system keeps the page cache in unused portions of the DRAM. Pages in the main memory that have been modified

during writing data to disk are marked as "dirty" and have to be flushed to disk before they can be freed. When a file write occurs, the cached page for the particular block is looked up. If it is not found in the page cache, the page(s) are fetched from disk and the requested modifications are done, read-on-write refers to unnecessarily fetching these pages .

To avoid this read-on-write issue, InnoDB checks whether the page that contains the target write data has already been in the page cache(first been written), if it's not, InnoDB will copy the contents to an 8KB Write Ahead Buffer(WA buffer), and fill up the remaining space of the buffer with 0s. Therefore, 8KB worth of pages will be allocated in the page cache at the beginning to prevent reads from occurring with the subsequent writes. Meanwhile, since InnoDB originally uses block devices to store data, the write granularity is the same as the block size used in OS to write data into block device, which is 512B. As a result, the InnoDB uses WA buffer in the following 2 situations, the first is when the log-write size is smaller than 512B. The other is when the log-write data is not aligned to 4KB page size, and the target area to be modified doesn't exist in the page cache.

Regarding to the previous two cases, InnoDB will copy the data to WA buffer and fills the remaining with 0s to align the size to 512B or 8K respectively. Particularly, the log_writer will handle the process after the log data has been written to the log buffer by making two critical copies. First, it will start by copying the log data to the Write-ahead (WA) buffer and memsets the rest of the space to zeros. After that, the log_writer will transfer-by calling pwrite()- the aligned data from the WA buffer to the page cache in the second copy, preventing the need for the OS to read the data from the device.

However, in our current environment, we are using optane DIMM by mounting it to the system in DAX mode. In this case, system page cache is being

bypassed when performing any I/O operations and immediately resolving read-on-write issues as we persist our modifications directly to the storage device. Therefore, by using the traditional write ahead mechanism during log write procedures, we believe the large alignment granularity in the WA buffer will result in extra unnecessary copies that contribute in evident performance degradation. On the other hand, as we mentioned in section 1.3.2, the device's physical access granularity is 256B. Any writes that are not aligned to this size will cause optane level read-on-write issue. For example, to serve a 128B write where the related data doesn't already exist in the optane hardware buffer, optane DIMM will read 256B to the buffer, modifies 128B, and writes the modified 256B resulting in amplified internal data reads to bring the whole block (256B) of data to the hardware buffer and then re-issue another write to the modifications.

NF-log aims to optimize the existing write ahead mechanism to exploit the features of optane DIMM. First, we started by deciding the new granularity and write ahead triggering threshold for log write. We cannot dispose of the whole write ahead mechanism, even though we could issue I/O requests to the optane DIMM in byte granularity, we mentioned earlier that in reality that that could result in read-on-write issues due to its coarse internal transfer granularity. According to the Intel optane DIMM's physical internal transfer size, the granularity and threshold should be 256B aligned, but too big granularity or threshold will introduce more overhead on data copy and page/block alignment. Therefore, after testing multiple possibilities from 256B to 4KB, we finally select 512B as our internal log write granularity and write ahead triggering threshold. We also conducted profiling analysis to gain more information on the log write size, we ran several benchmarks and real workloads and observed that at most of the time, the log write size is larger than 256B(between 256B and 512B). This indicates that by changing the granularity to 256B, we will witness incur

additional writes saturating the device bandwidth, so going this route is only possible if the log write requests are 256B or smaller all or most of the time, which is not the case. At the same time, granularity larger than 512B will incur additional overhead on copying data to the WA buffer.

Consequently, In NF-log's log_writer mechanism, we will only trigger the Write-Ahead mechanism if: each log write request is not 512B aligned or it does not already exist in the device's hardware buffer. In such conditions, the log_writer thread will copy the non-aligned part to the WA buffer, memset the rest of the block size with 0s and persist it directly to the optane DIMM from the log buffer.

Chapter 3

Evaluation

3.1 Experimental Setup

We evaluate our proposed NF-Log through synthetic and realistic workload. The environment is as following described:

plateform. We use 2-way Xeon Gold 5218 CPU that with 2.3GHz and 16-core per socket totaling to 32 physical and 64 logical cores with hyper-threading. It also has 160GB of RAM and two Intel optane DC Memory, each of which is 128GB totalling to 256GB. The Intel optane DC Memory in our evaluations is used in App Direct Mode, subsequently mounted with ext4 in Direct Access Mode(DAX), bypassing the system page cache.

Baseline and system configurations. Our original baseline is MySQL InnoDB(version 8.0), which also is the base of NF-Log implementations. Unless mentioned, all tests with InnoDB and NF-Log will share the following configurations. We set the write buffer to its default value and so is the log buffer. InnoDB is configured to use 1 thread for redo log writes to the page cache and 1

thread to flush to the storage device. It also uses 4 threads for data(and double write separately) writes(to page cache) and another 4 threads of asynchronous data write. All the remaining configurations are set to InnoDB default.

Workloads and datasets We use sysbench as a synthetic workload. Sysbench is a scriptable multi-threaded benchmark tool based on LuaJIT. It is most frequently used for database benchmarks. Sysbench can create many different workload patterns, especially write only and read write patterns are used in our evaluation. For sysbench, we use 32 tables each of them containing 10,000,000 rows, leading to the total size of all tables to 75GB. We vary the number of threads from 1 to 32 in order to observe the performance difference depending on different level of parallelism.

For real-world workloads, we use TPC-C [23] with MySQL. TPC-C involves a mix of five concurrent transactions of different types and complexity either executed on-line or queued for deferred execution. The database is comprised of nine types of tables with a wide range of records and population sizes. In our configuration, we load 2 different datasets, the first containing 100 warehouses and the second 750 warehouses to test the scalability of our system. We vary the number of connections from 1 to 32 to observe the performance difference on the different level of parallelism.

3.2 Performance breakdown

As we mentioned above, NF-log contains 3 different optimizations: immediate persist, write ahead optimization and data access locality optimization. In this section, we break down the performance, and analyze how much improvement can be achieved due to each optimization.

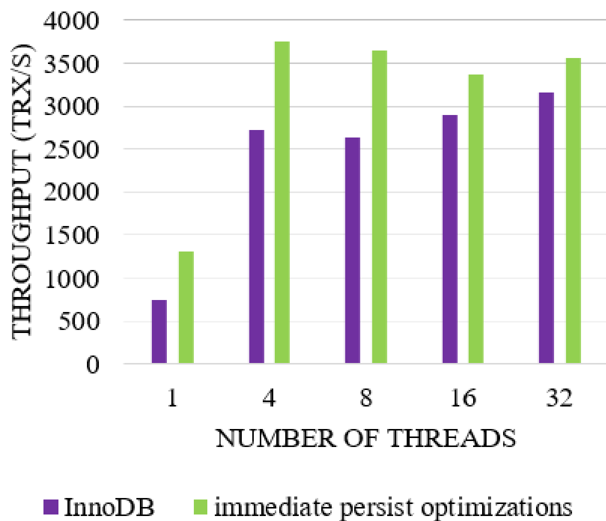


Figure 3.1 Evaluating immediate persist optimization with sysbench write only workload

3.2.1 Evaluating immediate persist optimization

We first analyze the performance improvement achieved by immediately persisting the log writes to the device from the log buffer. In this evaluation, we isolated this specific optimization of NF-Log and kept the rest to its default state. We ran sysbench oltp_write_only varying the number of user threads and compared the results with InnoDB baseline. Figure 3.1 demonstrates the results of this evaluation. According to figure 3.1, the immediate persists optimization can achieve up to 38% improved throughput compared the original InnoDB when running four threads. this achievement is mainly due to disabling redundant persists in InnoDB. However, the improvement decreased to 13% when using 32 threads. Since we removed system overhead, the performance is supposed to reflect the internal features of optane DIMM more directly, which is optane DIMM does not having a high internal parallelism. [24] We can clearly visualize this fact after running InnoDB directly on the device with minimal

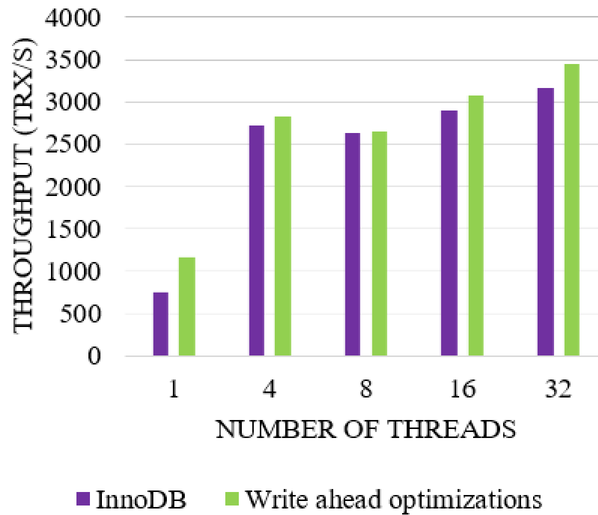


Figure 3.2 Evaluating write ahead optimization with sysbench write only workload

system interference.

3.2.2 Evaluating write ahead optimization

In this section, we verify our design regarding to write ahead related optimizations. We measured the performance by running Sysbench oltp_write_only workload. In this evaluation, we disabled immediate persists and data access locality optimizations to isolate the write-ahead optimization performance gain. Figure 3.2 shows the result with different levels of parallelism. We can observe that the optimization of write ahead mechanism outperforms original InnoDB by up to 9.3% when dispatching 32 threads. However, the improvement is not significant enough for the other cases. This is because for write ahead optimization, we basically remove the overhead of extra memory copy to align a 8KB sized write-ahead buffer and choose to use a 512B sized write ahead buffer to fit the optane DIMM internal access granularity. The performance gain will

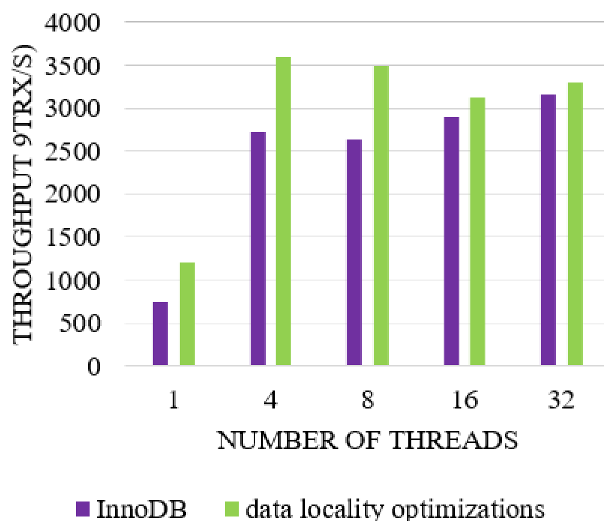


Figure 3.3 Evaluating data access locality optimization with sysbench write only workload

only occur when the address of write contents is not located in the WA buffer area. To visualize better performance, it is best to dispatch a larger number of running threads. This relates to the fact that more running threads trigger more WA copies, and the more copies to the WA buffer, the higher possibility of the write contents address not being in the WA, therefore highlighting its performance.

3.2.3 Evaluating data access locality optimization

In this section, we analyze the data access locality optimization. Similar to the previous two experiments, we use sysbench write_only_workload to conduct the evaluation. We isolated this optimization from other two on the system and compared it to the original InnoDB. According to figure 1.3.2, data locality optimization can bring 31.8% and 32.5% throughput improvement with 4 and 8 threads respectively. However, with the parallelism increasing to 32, the im-

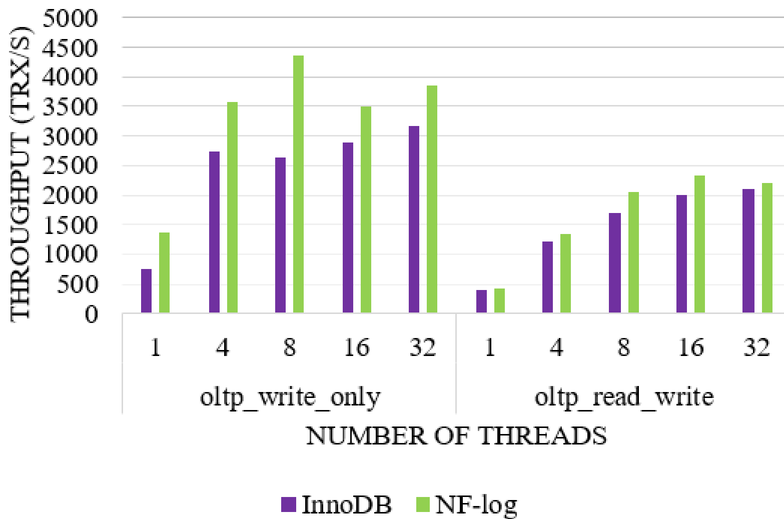


Figure 3.4 Sysbench overall performance

provement range shrinks to 4%. We believe this is caused by the following 2 reasons. First, The Intel optane DIMM has a small internal parallelism and increasing number of outstanding requests will cause internal congestion. Second, we implement data access locality optimization by pinning I/O threads to one of the CPU sockets installed on the machine. Therefore, dispatching a large number of threads will break the load balancing and cause additional overhead on the CPU socket we are operating on. In our future work, we will consider to dynamically balance the load while preserving data access locality.

3.3 Overall Performance

Here we use Systbench and TPC-C to evaluate NF-Log's overall performance in comparison to InnoDB.

Sysbench write-intensive workload (oltp_write_only) After running the benchmark for 60 seconds multiple times and calculating the average. Under

these conditions InnoDB(non-optimized) delivers uniformly lower performance across all configurations(thread number variations). This reveals how existing InnoDB's sequential logging via file system, fail to utilize the features of Intel's optane DC memory. From these results in Figure 3.4, NF-log improves the throughput in transactions per second by 1.65x for the same configuration, which is when dispatching 8 threads and by 1.38x across all configurations, performing 1204 more transactions per second over InnoDB. This is mainly due to closing the bridge between the log buffer and the disk by performing immediate persists using PMDK, utilizing the finer-grained addressability of the device and alleviating the unnecessary impact of the page cache on the system. In addition to that, with a smaller granularity and write ahead mechanism triggering threshold, we no longer handle large size data copy but 512B aligned requests which coincides with the internal transfer granularity of the Intel optane persistent Memory. Both these optimizations coupled with improved data locality we were able achieve such boosted performance.

Sysbench read-write workload (oltp_read_write) As NF-Log mainly targets write optimizations, specifically the redo log part, we expect smaller improvements with read_intensive and almost non-existent improvement with read_only workloads. Sysbench oltp_read_write is a mix workload but it is classified as read-intensive as it contains 75% reads and 25% writes. As shown in figure 3.4, NF-log will reach its maximum performance at 16 threads with 16% improvement compared to InnoDB running the same configuration, and 9% total improvement across all configurations.

Sysbench read-intensive workload (oltp_read_only)

Even though performing optimizations on the redo log should not affect the read performance as it is not going to be written other than at recovery time. We still ran read_only workloads to verify that in fact it has not been impacted.

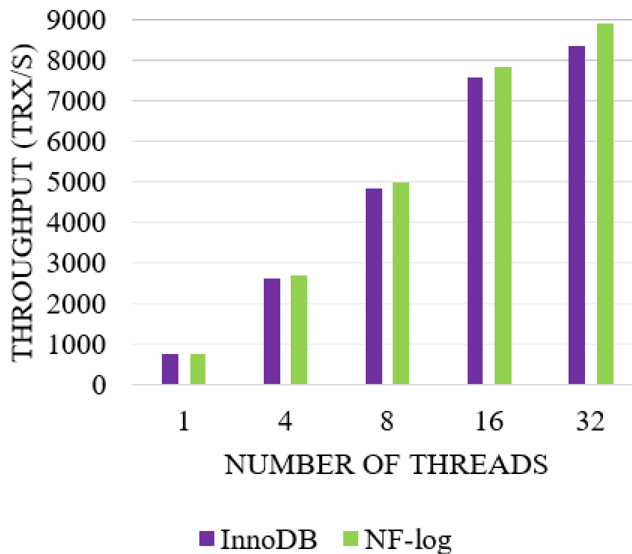


Figure 3.5 Sysbench Read Intensive Performance

We can see in figure 3.5, the throughput from running 1 million transactions. We can see that NF-log and InnoDB still perform very similarly, but we can see that NF-log is 3% better.

TPC-C real-world workload TPC-C workload is originally mixed with a ratio of (75% - 25%) read compared to write. Out of five transactions performed, SLEV(Stock-Level) And OSTAT(Order-Status) represent heavy read-only database transactions, the other three are NEWORD(New-Order), DELIVERY and PAYMENT are read-write executions with weight variations. We start by running the original TPC-C with 2.8% NEWORD, 2.8% PAYMENT, 2.8% OSTAT, 45% DELIVERY and 45% SLEV. We ran the evaluation on 100 warehouses totaling the dataset to 10GB and then 750 warehouses matching our previous experiments with 75GB to test our system's data scalability. Before measuring the performance, we set the ramp-up time to 100seconds, and then measure the performance for 300 seconds. Presented are the average results af-

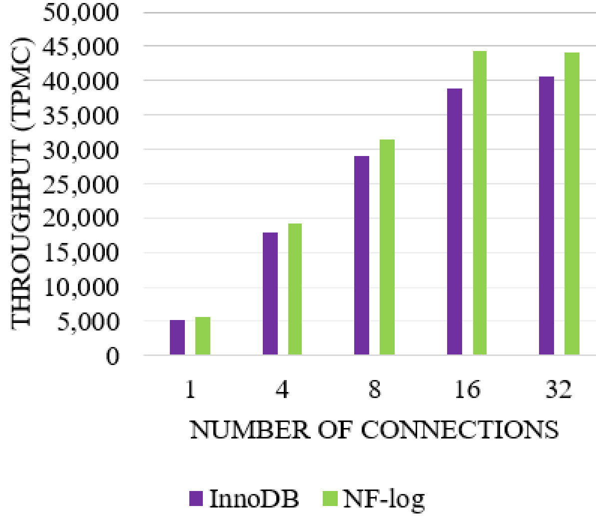


Figure 3.6 Peformance with TPC-C Bechmarks, 100W, NEWORD(2.8%), PAYMENT(2.8%), OSTAT(2.8%), DELIVERY(45%), SLEV(45%)

ter several rounds of measurements. As shown in figure 3.6, the results were similar to the previous workload since our modification is happening mainly on the write path. We can see a slightly smaller gain than yielded by write-intensive workloads. We noticed that under real-world workloads, NF-log reaches the saturation at 16 threads similar to the synthetic mixed workload. However, Unlike in sysbench oltp_write_only workloads, and matching the mixed ones, it yields an overall performance gain of 1.12x over InnoDB.

TPC-C real-world workload To better visualize our optimization, we performed modifications on the run-time of certain transactions in TPC-C. this time we are running 5.5% NEWORD, 5.5% PAYMENT, 0% OSTAT, 89% DELIVERY and 0% SLEV, by eliminating the heavy read_only transactions we are able to run the evaluations in a more write_intensive environment. As expected, we see an improvement with the increase of update ratio. For 10GB of data we can observe a maximum improvement of 24% with 32 connections

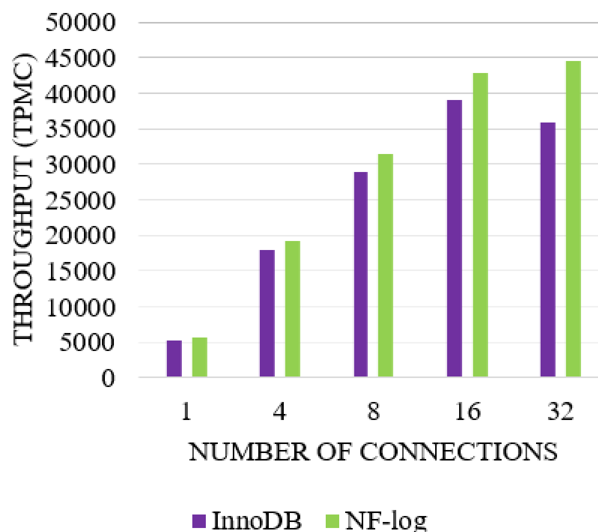


Figure 3.7 Peformance with TPC-C Bechmarks, 100W, NEWORD(5.5%), PAYMENT(5.5%), OSTAT(0%), DELIVERY(89%), SLEV(0%)

and a global best to best improvement of 13% as show in figure 3.7. In the case of 75GB dataset, the improvement is slightly smaller as NF-log achieves 1.16x the performance of InnoDB over all configurations and a maximum of 1.18x for 32 connections demonstrated in figure 3.8.

Among all tests, NF-log demonstrates its cost effectiveness and excellent utilization of the device in write intensive conditions, it is able to highlight the benefit of in-memory databases and encourages on further optimizations on other parts of the RDBMS.

3.4 Resource utilization

We wanted to confirm the performance of our system by comparing the resource utilization.

Persistent memory. We start with the persistent memory. We used pcm.x

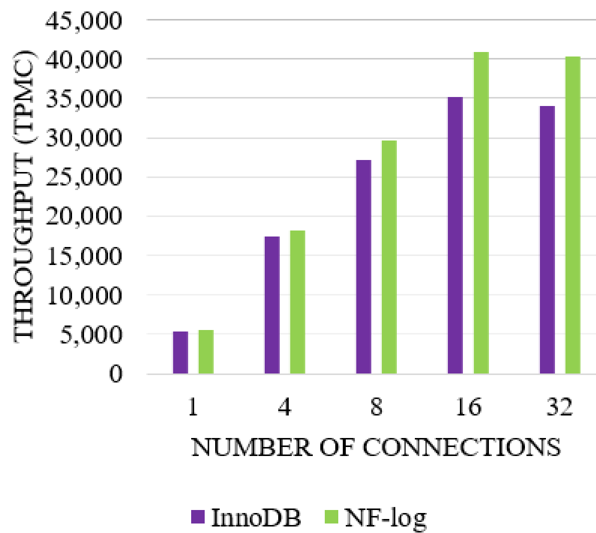


Figure 3.8 Peformance with TPC-C Bechmarks, 750W, NEWORD(5.5%), PAYMENT(5.5%), OSTAT(0%), DELIVERY(89%), SLEV(0%)

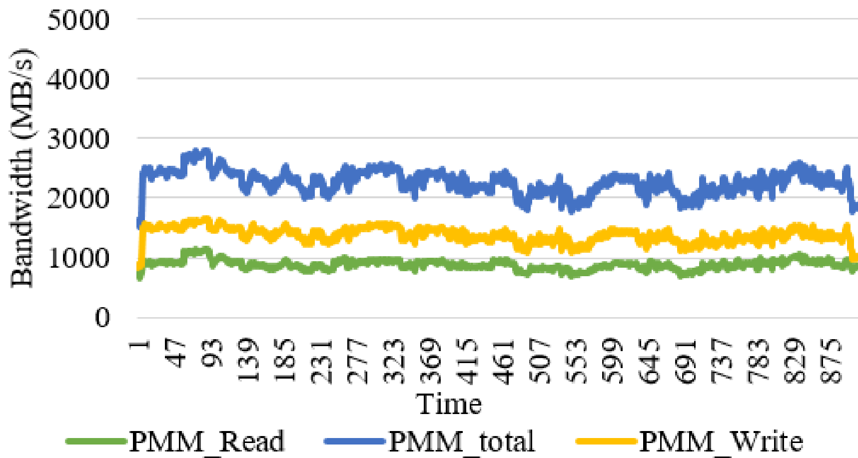


Figure 3.9 InnoDB 1million write transactions PMEM utilization

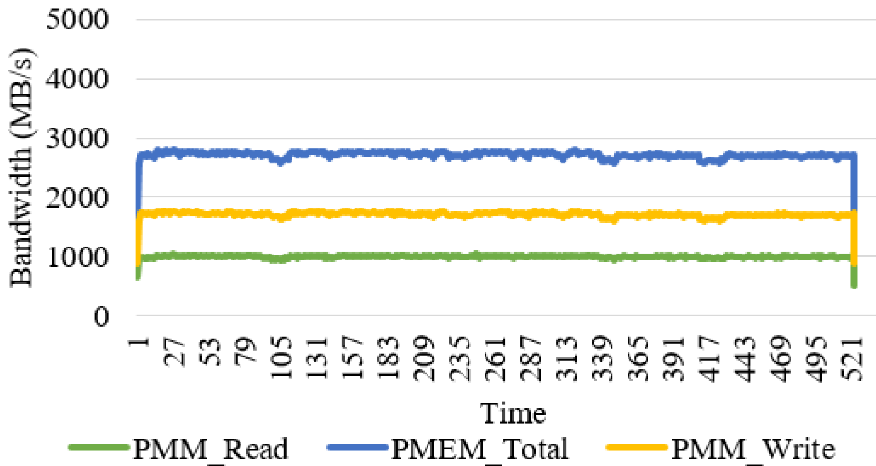


Figure 3.10 NF-log 1million write transactions PMEM utilization

to track the memory bandwidth used while running sysbench oltp_write_only. We ran 1 million write transactions resulting in 4 million write queries. All of these values were measured pcm.x every 1 second, in figures 3.9, 3.10 the X axis represents the execution time 1 second at a time and the Y axis represents the correspondent persistent memory bandwidth. The DB is the only running task during the time of the experiment.

The histograms represented in the figures mentioned above depict separately the read and write PMEM bandwidth and the green total represents the sum of both of them. According to both figures 3.9 and 3.10 we can notice that NF-log has a more stable performance across the time of measurement. We saw in past performance experimental results that NF-log achieves 44% speedup for 8 threads but in this context it only uses 19% more resources.

DRAM occupied size and bandwidth. For DRAM on the other hand, by running the same experiment in the previous paragraph and measuring the memory bandwidth every 1 second. we can expect 15% better dram bandwidth utilization on top of stable performance due to eliminating page cache overhead.

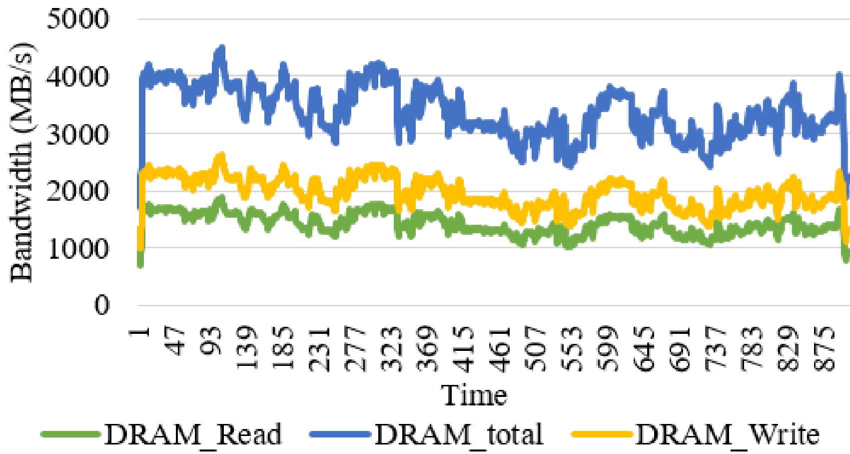


Figure 3.11 InnoDB 1million write transactions DRAM utilization

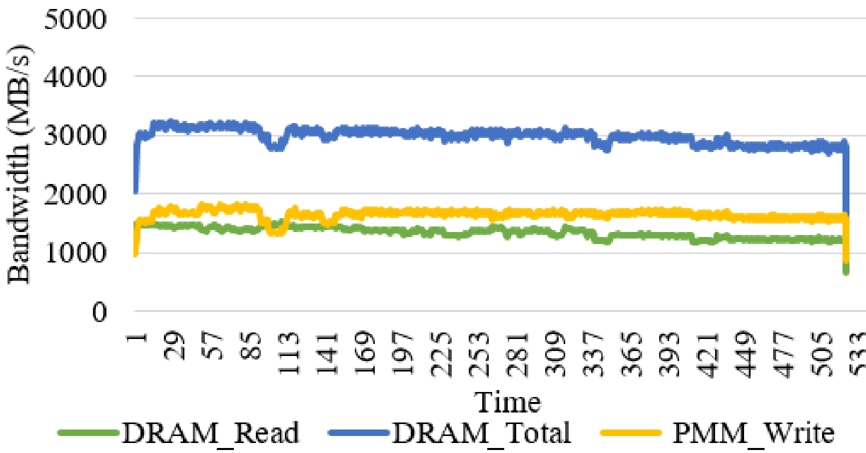


Figure 3.12 NF-log 1million write transactions DRAM utilization

The size of resident memory(the size the program occupies in the DRAM) is very similar. When InnoDB uses 593284 KB in the memory, InnoDB uses 591880 KB, therefore not introducing any overhead.

CPU. Under the same experimental conditions, we measured the CPU utilization of reach the original and optimized systems. The results show that with NF-log we needed less resources to achieve an even better throughput. We ran sysbench oltp_write_only and as NF-log completed of the transactions completed by InnoDB. It utilized 1136% of CPU. Similarly, InnoDB used up 1173% of CPU, both averaging to 11 cores fully saturated across the duration of the experiment. Such results further demonstrate that NF-log will better utilize PMEM environments and still not induce any overhead on other resources.

3.5 Write Size

The final evaluations in our paper is testing weather NF-log actually reduces the number of writes from removing the redundant persists. We ran sysbench oltp_write_only for 1 million write transactions and used pcm.x to measure the size of data written to the Optane DC memory during the total period of the experiment.

As we can see in figure 3.13, by performing Immediate persists we see a 30% reduction in the write size to the device. From seeing these results, we can say that NF-log demonstrates its cost effectiveness and excellent utilization of the device in write intensive conditions, it is also able to highlight the benefit of in-memory databases and encourages on further optimizations on other parts of RDBMS.

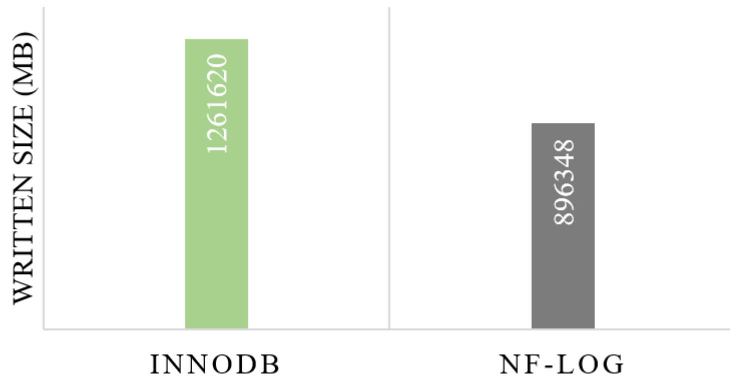


Figure 3.13 InnoDB 1million write transactions DRAM utilization

Chapter 4

Conclusion

In this paper we presented NF-Log, which revisits the redo log component of InnoDB to better utilize the high performance and low latency of optane DC memory as a storage device. The redesign includes several re-considerations to the synchronizations among the threads responsible for writing, flushing and checkpointing the redo log. Due to bypassing the page cache, we invalidated the need to separate the log writer and flusher, as this separation is meant for handling one more step involving system page cache. Therefore we chose to merge the functionality of log_flusher into the log_writer to persist log data to the optane DIMM immediately from the log buffer. We also modified the write ahead mechanism by removing the extra copies using the write_ahead buffer solely for aligning blocks to the desired size. Finally, we improved persistent memory data locality after observing that remote persistent memory access is causing significant overhead to the overall system. The evaluation shows that NF-Log can boost the performance up to 39% regarding to write intensive workload when using Sysbench. For the real-world workload such as TPC-C, NF-Log can

still outperform the original InnoDB by 16% even though the workload is read intensive. The system demonstrates better resource utilization and reduces the write size by 30%

Bibliography

[1] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, p. 1537–1550, may 2016.

[2] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane dc persistent memory module," 2019.

[3] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+redo logging for persistent memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 336–349, 2018.

[4] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the nvram era," *Proc. VLDB Endow.*, vol. 7, p. 121–132, oct 2013.

[5] T.-D. Nguyen, J.-H. Park, and S. W. Lee, "Revisiting write-ahead logging with nvdimm," 10 2019.

[6] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus, "Fine-grain checkpointing with in-cache-line logging," in *Proceedings of the Twenty-Fourth Inter-*

national Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, (New York, NY, USA), p. 441–454, Association for Computing Machinery, 2019.

[7] N. Katzburg, A. Golander, and S. Weiss, "Nvdimm-n persistent memory and its impact on two relational databases," in *2018 IEEE International Conference on the Science of Electrical Engineering in Israel (ICSEE)*, pp. 1–5, 2018.

[8] Y. Qiao, X. Chen, J. Hao, T. Zhang, C. Xie, and F. Wu, "Architecting heterogeneous memory systems with dram technology only: A case study on relational database," in *2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pp. 25–33, 2020.

[9] J. Yang, B. Li, and D. J. Lilja, "Heuristicdb: A hybrid storage database system using a non-volatile memory block device," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[10] P. Frühwirt, P. Kieseberg, S. Schrittwieser, M. Huber, and E. Weippl, "Innodb database forensics: Reconstructing data manipulation queries from redo logs," in *2012 Seventh International Conference on Availability, Reliability and Security*, pp. 625–633, 2012.

[11] "pmdk." `https://pmem.io/pmdk/`.

[12] H. Jung, H. Han, and S. Kang, "Scalable database logging for multicores," *Proc. VLDB Endow.*, vol. 11, p. 135–148, oct 2017.

[13] J. Arulraj, M. Perron, and A. Pavlo, "Write-behind logging," *Proc. VLDB Endow.*, vol. 10, p. 337–348, nov 2016.

[14] J. A. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. B. Zdonik, and S. R. Dulloor, "A prolegomenon on oltp database systems for non-volatile memory," in *ADMS@VLDB*, 2014.

[15] D. Koutsoukos, R. Bhartia, A. Klimovic, and G. Alonso, "How to use persistent memory in your database," 2021.

[16] H. Avni, A. Avitzur, N. Pachter, and V. Vexler, "Extending in-memory oltp with persistent memory," 08 2021.

[17] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "Chameleondb: A key-value store for optane persistent memory," in *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, (New York, NY, USA), p. 194–209, Association for Computing Machinery, 2021.

[18] 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*, DaMoN '14, (New York, NY, USA), Association for Computing Machinery, 2014.

[19] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 520–532, 2018.

[20] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane dc persistent memory module," 2019.

[21] "quick start guide." 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.

[22] "mysql." https://dev.mysql.com/doc/dev/mysql-server/latest/.

[23] "tpcc." https://www.tpc.org/tpcc/.

[24] J. Liu, S. Chen, and L. Wang, "Lb+trees: Optimizing persistent index performance on 3dxpoint memory," *Proc. VLDB Endow.*, vol. 13, p. 1078–1090, mar 2020.

[25] "azure." https://azure.microsoft.com/en-us/products/azure-sql/database/#overview.

[26] "oracle." https://www.oracle.com/cloud/.

[27] "sap hana." https://www.sap.com/products/hana/cloud.html.

요약

비휘발성 메모리는 새로운 스토리지 기술로 고성능과 바이트 주소 접근 특성을 연결했지만 기존 관계형 데이터 베이스에 적용이 많이 진행되어 있지 못하고 있다. 그것은 기존 관계형 데이터 베이스들이 데이터와 로그를 블록 저장장치에 저장하는 걸 가정하고 설계되기 때문이다. 결과적으로 디비에서 발생하는 쓰기 작업은 비휘발성 메모리를 충분히 사용 못하며 전체 성능을 저하할 수 있다. 본 연구는 우선 InnDB에서 발생하는 redo 로그에 대해 전반적으로 분석하여 해당 로그에 발생하는 작업은 전반적인 성능에 대한 영향을 분석했다. 그것은 바탕으로 NF-Log를 제시했다. NF-Log는 기존의 로그의 중복적인 플러시 현상을 제거하고 페이지 캐시에 대한 영향도 최소화 했다. 또한 본 연구에서 원격 비휘발성 메모리에 적근할 때 발생하는 오브헤드를 최소화 했고 미리 쓰기 버퍼에 대한 최적화도 했다. 결과적으로 본 연구는 비휘발성 메모리에 바이트 접근성과 영구성 특성을 이용함으로 기준 InnoDB보다 쓰기 사이즈를 30%를 줄였고 sysbench인 경우 성능을 38% 증가시켰고 TPC-C인 경우 16%를 증가했다.

주요어: 비휘발성 메모리 환경에서의 관계형 데이터베이스 시스템 연구
학번: 2020-28751

Remerciments

Mes vifs sincères remerciements s'adressent particulièrement à :

- **Prof Heon Young Yeom**, Mon encadrant à l'**Université Nationale de Seoul – SNU**, pour m'avoir prodigué l'honneur de travailler dans son équipe. Je tiens à exprimer toute ma gratitude pour l'aide qu'il m'a apporté durant toutes les phases de cette formation. Sa disponibilité, son encadrement, sa pédagogie et ses conseils m'ont été précieux pour atteindre les objectifs de ce projet dans les délais convenus.

- Mon mentor **Dr Qichen Chen** pour la qualité de son encadrement, son encouragement et ses conseils sans oublier sa participation au cheminement de ce projet.

- Aux rapporteur et président de jury pour l'honneur qu'ils m'ont fait de participer à l'examen de mon travail.

- Mes amis au **Laboratoire des Systemes Distribues – DCSLAB** pour la disponibilité, la gentillesse et la sympathie dont ils ont fait preuve tout au long des deux ans écoulés et m'ont permis de poursuivre ce projet dans les meilleures conditions.

- Tous les enseignants qui ont participé à mon évolution scientifique durant les deux années de ma formation à l'**Université Nationale de Seoul – SNU**.

- A toute personne qui a contribué à l'élaboration de ce travail.

Dedicace

Je dédie ce travail

- A ma mère **BEN SAIDA Habiba**, pour son amour, ses encouragements et sacrifices.

- A mon père **ZGUEM Semy**, pour son soutien, son affection et la confiance qu'il m'a accordé.

- A mes frères et scoeurs, **Aziz, Salim, Iyed, Asma, Omayma, Senda**

- A mes grands supporteurs dans la vie et lors de mon carriere et formation scholaire **BEN CHEIKH Hayet et Yahia**

- Mes amis que j'ai rencontré lors de mon expérience avec GKS

- A tous ce qui m'aiment