Evaluating a Trade-Off between DRAM and Persistent Memory for Persistent-Data Placement on Hybrid Main Memory

Satoshi Imamura
Fujitsu Laboratories Ltd.
Kanagawa, Japan
s-imamura@jp.fujitsu.com

Mitsuru Sato
Fujitsu Laboratories Ltd.
Kanagawa, Japan
msato@jp.fujitsu.com

Eiji Yoshida
Fujitsu Laboratories Ltd.
Kanagawa, Japan
yoshida.eiji-01@jp.fujitsu.com

ABSTRACT

Persistent memory will be widely used as main memory in future computer systems because it is byte addressable, its cost is lower than that of DRAM, and its access speed is much higher than that of secondary storages. However, since the access speed (especially write speed) and write endurance are lower than those of DRAM, it will be combined with DRAM to constitute main memory. This is well-known as *hybrid main memory system*, and various techniques to optimize data placement on this system have been proposed. The fundamental concept of them is to place frequently written data on DRAM in order to hide the disadvantage of persistent memory.

In this paper, we focus on applications which guarantee data persistence with logging mechanisms and suggest a new trade-off between DRAM and persistent memory on platforms that apply hybrid main memory with secondary storages. If data is placed on volatile DRAM, it can be accessed faster but logging incurs a non-trivial overhead. On the other hand, if the data is placed on non-volatile persistent memory, it suffers from the slower accesses while it can avoid the logging overhead. We quantitatively evaluate this trade-off using an in-memory database as a case study on a persistent memory emulation platform. The results reveal that it is better for performance to place frequently written data on persistent memory rather than DRAM. Compared to placing the data of a workload including 50% reads and 50% writes on DRAM with logging enabled, we can achieve 38% and 25% higher throughput of the database by placing the data on PM whose latency is double and four times of the DRAM latency, respectively.

CCS CONCEPTS

• **Information systems** → **Data layout**; • **Hardware** → **Memory and dense storage**;

KEYWORDS

Hybrid main memory systems, DRAM, persistent memory, data placement, logging mechanisms

ACM Reference Format:
Satoshi Imamura, Mitsuru Sato, and Eiji Yoshida. 2017. Evaluating a Trade-Off between DRAM and Persistent Memory for Persistent-Data Placement on Hybrid Main Memory. In *Proceedings of Workshop on Hardware/Software Techniques for Minimizing Data Movement, Portland, Oregon, USA, September 2017 (Min-Move '17),* 6 pages.

1 INTRODUCTION

Emerging persistent memory (PM) is currently attracting a lot of attention both in industry and academia, because it has excellent properties in terms of performance and cost. It provides five times the capacity at the same cost as DRAM [5] and three orders of magnitude lower the read latency as storages [22]. More importantly, PM is byte addressable and non-volatile, which means that data stored on PM is not lost across system power failures. Real PM products will be available in near future because Intel has announced that 3D XPoint-based PM will be released in 2018 [7].

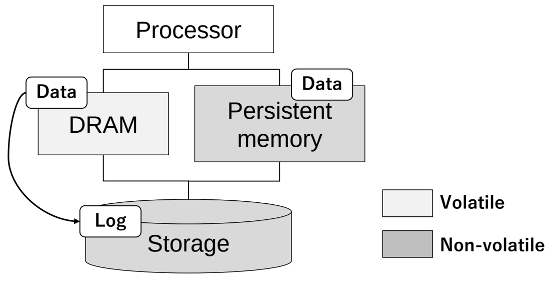

Figure 1: Data placement on a platform including hybrid main memory and a storage for an application which guarantees data persistent with logging.

However, the access speed (especially write speed) and write endurance of PM are lower than those of DRAM. If data stored on PM is frequently written, it suffers from the long access times and the cells may be worn out in a short time. In order to cover these disadvantages of PM, *hybrid main memory* including DRAM and PM will be commonly applied to future computer systems. As PM can be accessed with common loads and stores via the same interface to DRAM, it can be combined with DRAM to constitute a single physical address space of main memory. For optimization of data placement between DRAM and PM on hybrid main memory, several hardware- and software-based techniques have been proposed [11, 12, 14, 19]. The fundamental concept of them is to reduce the number of writes to PM by placing frequently written data on DRAM. However, their viewpoints are only the differences of the write speed and endurance between DRAM and PM.

In this paper, we target applications which guarantee data persistence, such as in-memory databases, and focus on the difference of volatility between DRAM and PM. Such applications traditionally apply logging mechanisms to record the information about each data update in a log file on non-volatile storages. If data on volatile DRAM is lost due to system failures, the log is used to recover the data. In contrast, logging is not necessary if the data is persisted on non-volatile PM. As logging incurs a non-trivial performance overhead [13], a new trade-off between DRAM and PM must be taken into account to maximize the performance of such applications on platforms that apply hybrid main memory with storages, as illustrated in Figure 1. If data is placed on DRAM, it can be accessed faster but the logging overhead occurs to guarantee data persistence. On the other hand, if the data is place on PM, it suffers from the slower accesses while it can avoid the logging overhead.

We choose an open-source in-memory database, Tarantool [21], as a case study and quantitatively evaluate the trade-off using a platform provided by Intel that can emulate the latency and bandwidth of PM. The logging overhead occurred by placing data on DRAM is evaluated by running Tarantool on DRAM with logging enabled and disabled. The overhead of slower accesses occurred by placing data on PM is evaluated by running Tarantool on the emulated PM without logging. In order to guarantee data persistence and transaction atomicity on PM, we modify the source code of Tarantool using the *libpmemobj* library of NVM Library [18].

Our evaluation shows that the logging overhead becomes larger than the overhead of slower accesses to PM, when data is frequency written. This result means that it is better for performance to place frequently written data on PM rather than DRAM. For example, compared to placing the data of a workload including 50% reads and 50% writes on DRAM with logging enabled, we can achieve 38% and 25% higher throughput of Tarantool by placing data on PM whose latency is double and four times of the DRAM latency, respectively. This is an opposite finding to the concept used in existing data placement techniques for hybrid main memory. Therefore, we need to rethink the data placement if our objective is to maximize the performance of applications which guarantee data persistent.

This paper is organized as follows. Section 2 introduces prior work related to this work and discusses the difference between them. Section 3 explains the architecture of our target computer system and our experimental platform for PM emulation. Section 4 describes an in-memory database as a case study of applications which guarantee data persistent and suggests the new trade-off on hybrid main memory. Section 5 provides the methodology and results of our evaluation. Finally, we conclude this work in Section 6.

2 RELATED WORK

In order to cover the disadvantages of PM, applications have been optimized at an algorithm- and data structure-level. Chen et al. designed two PM-friendly database algorithms: B^+-tree index and hash joins [2]. They re-organized the data structures to reduce the number of writes at the expense of an increase in reads, thereby achieving higher overall performance and endurance. Bailey et al. presented a persistent key-value storage system designed to leverage the advantages of PM [1]. Their system relies on byte-addressability of PM to support both fine- and large-grained data managements and employs a two-level hierarchical design. It can offset the performance and endurance weakness of PM by managing thread-local data on DRAM,

Oukid et al. implemented a database that efficiently works on hybrid main memory without storages [16, 17]. As it directly updates primary data on PM, there is no need to copy data from storages. More importantly, by leveraging non-volatile PM, the logging mechanism of the database can be dropped and data on PM can be instantly recovered after system failures. Index data structures can be placed on either DRAM or PM to trade-off the overall throughput and data recovery times. Compared to their work, we assume platforms that apply hybrid main memory with storages (as illustrated in Figure 1), because the capacity of PM is still much smaller than that of storages. As primary data can be also placed on either DRAM or PM on such systems, the trade-off described in Section 1 must be taken into account to maximize the performance of applications.

A traditional logging mechanism of databases, called *write-ahead logging* (or *physical logging*), records the information about each data update including before and after images to non-volatile storages [15]. This logging mechanism represents a non-trivial fraction of the overall transaction execution time; therefore, it is a big challenge to reduce this overhead for improving the throughput of databases. Malviya et al. proposed a light-weight, coarse-grained logging technique called *command logging*, which only records the executed transactions instead of each data update [13]. Command logging can significantly reduce the logging overhead compared to physical logging, but recovery times after failures become longer because transactions need to be re-executed completely. In contrast, we aim to eliminate the overhead of write-ahead logging by placing data on non-volatile PM while keeping short recovery times. On the other hand, PM has been used to store logs for a reduction in the logging overhead [6, 8]. This approach can avoid writing logs to slow storages and eliminate the impact of storage I/O time on performance. Placing data itself on non-volatile PM, as considered in this paper, is an orthogonal approach to the PM-based logging because it aims to completely eliminate the process of logging. Its advantage is to save the capacity of PM for log entries.

In order to efficiently exploit the potential of hybrid main memory, several techniques to optimize data placement between DRAM and PM have been proposed. As mentioned in Section 1, the fundamental concept of them is to place frequently written data on DRAM to avoid frequent writes to PM. Mogul et al. showed that operating systems can know the write frequency of pages, and the such information is useful to appropriately manage pages [14]. Ramos et al. implemented a memory controller to monitor access patterns, migrate pages between DRAM and PM, and translate the memory requests from cores [19]. Li et al. analyzed the workload characteristics of several large-scale scientific applications and showed that rigorously managing write accesses is essential to improve the performance and reduce the power consumption [12]. Lee et al. proposed an operating system-level page management algorithm that uses the write frequency of each page as well as the recency of write references to accurately estimate future write references [11]. Dulloor et al. showed that memory access patterns such as sequential, random, and pointer chasing are important to decide data placement and designed a set of libraries and automatic tools that enables programmers to achieve optimal data placement [5]. Compared to the above prior work, this paper reveals that it is better for performance to place frequently written data on PM rather than DRAM for applications which guarantee data persistence.

3 ARCHITECTURE OF TARGET SYSTEM

This section describes persistent memory, the architecture of our target computer systems, and our experimental platform to emulate persistent memory.

Table 1: Comparison of Memory Technologies [2, 22].

	DRAM	PM (PCM)	Flash	HDD
Cell size (F^2)	6-12	4-16	1-4	N/A
Read latency	20-50 ns	48-70 ns	< 25 us	< 5 ms
Write latency	20-50 ns	< 1 us	< 500 us	< 5 ms
Write BW (MB/s)	1000	50-100	5-40	< 200
Endurance	$> 10^{16}$	10^9	10^4	$> 10^{16}$
Byte addressable	Yes	Yes	No	No
Non-volatile	No	Yes	Yes	Yes

3.1 Persistent Memory

PM is byte-addressable and non-volatile memory, which can be accessed with common loads and stores via the same interface to conventional DRAM [3]. It is implemented as dual inline memory modules (DIMMs) and attached to a memory bus side-by-side with DRAM. Although several technologies such as spin-torque transfer magnetoresistive RAM (STT-MRAM) and ferroelectric RAM (FeRAM) have been considered as candidates for PM, phase-change memory (PCM) is known as one of the most promising technologies [10]. In this paper, we assume PM implemented based on PCM.

Table 1 compares the characteristic of PCM-based PM with those of DRAM, Flash solid state drives (SSDs), and hard disk drives (HDDs). The cell size of PM is comparable to that of DRAM but larger than that of Flash SSDs. Thus, it is still not practical to completely substitute PM for Flash SSDs. The read latency of PM is three and five orders of magnitude lower than those of Flash SSDs and HDDs, and PM exposes over two times higher endurance than Flash SSD. Therefore, PM is expected to be used as main memory. However, compared to DRAM, the disadvantages of PM are the slower access speed and lower write endurance. Although the difference of read latency is not large, the write latency and bandwidth is much worse. It is a big challenge to hide these disadvantages if PM is used as main memory.

3.2 Hybrid Main Memory with Storages

With the characteristic of PM explained in the previous section, PM will be combined with DRAM to constitute main memory in future computer systems, which is called hybrid main memory. DRAM can be leveraged to compensate the lower write speed and endurance of PM. In addition, we assume that secondary storages are attached to hybrid main memory, as illustrated in Figure 1, because the capacity of PM is still much smaller than that of storages. On such systems, we need to appropriately place data either on DRAM or PM in order to fully exploit the potential of hybrid main memory.

3.3 Emulation Platform

Since PM is not yet commonly available, Intel provided a platform to emulate PM [9]. Table 2 summarizes the specification of our experimental platform. The emulation of PM is based on the support of a Linux kernel and two separate DRAM regions (64 GB region on channels 0-1 and 384 GB region on channels 2-3 of this platform). The latter region is reserved by modifying kernel command line parameters (like "memmap=nn!ss"), and it appears as a PM region to Linux. The accesses to the emulated PM region is provided by a kernel module, which exposes the region as a block device. This block device is formatted with an ext4 file system and mounted with the direct access (DAX) extensions enabled. The DAX support of ext4 allows direct load/store accesses at byte granularity to the contents of the files in this file system. As the PM region is exposed by Linux as memory-mapped files, applications can create memory pools from the files.

Table 2: Specification of our platform for persistent memory emulation.

CPU	Intel Xeon E5-4620 v2, 2.6 GHz, 8 cores
Memory	64 GB DDR3 (channels 0-1) DRAM latency: 85 ns, DRAM bandwidth: 38 GB/s 384 GB DDR3 (channels 2-3) for PM emulation PM latency: 170 ns (2x), 340 ns (4x), 680 ns (8x) PM bandwidth: 9.5 GB/s (1/4)
HDD	1 TB SATA
OS	Linux kernel 4.7.0-3.el6.FTS

Moreover, this platform has special machine specific registers (MSRs) to configure the latency and bandwidth of PM. Based on the register values, this platform injects additional stall cycles into all memory accesses and limits the memory bandwidth. We change the latency to 170 ns, 340 ns, and 680 ns, which correspond to 2x, 4x, and 8x of the DRAM latency, respectively. In addition, we set the PM bandwidth to a quarter of the DRAM bandwidth (this setting is only available on our platform). Note that these settings affect both reads and writes to the PM region. These are pessimistic settings because the actual read latency of PM is lower than the write latency.

4 CASE STUDY: IN-MEMORY DATABASE

In this work, we choose an in-memory database, Tarantool [21], as a case study of applications which guarantee data persistence with logging. This section presents the overview of this database and a new trade-off on hybrid main memory.

4.1 Tarantool

Tarantool is an open-source NoSQL in-memory database management system. It processes transactions on a single thread, thereby guaranteeing consistency and isolation. Moreover, all data is maintained in main memory, and atomicity and durability (or data persistence) are ensured by a common write-ahead log (WAL) [15] and a snapshot stored in a non-volatile storage. Tarantool creates an entry that contains the information about each data update (e.g., before and after data images) and records it in the log. In addition, it takes a snapshot of all data on DRAM and deletes the old log at a certain interval. If data on DRAM is lost due to a system failure, Tarantool can recover them by reading the latest snapshot and replaying the transactions that are recorded in the log. The write-ahead logging can be enabled or disabled with an execution parameter. Tarantool supports four types of data structures for index (hash, tree, bitset,

Table 3: Pros and cons of DRAM and persistent memory for applications which guarantee data persistence.

	Access speed	Logging
DRAM	Faster	Necessary
PM	Slower	Unnecessary

and rtree), and we use the tree data structure that is the default setting.

4.2 Data Management on Persistent Memory

When Tarantool writes data to PM, the write is processed on processor caches at first; therefore, the data must be persisted by flushing the corresponding cache line to PM. Intel recommends the use of an optimized cache flushing instruction named *CLWB* [20]. As this instruction does not invalidate a cache line when flushed, the cache line can be accessed immediately by following instructions. Although the flushed cache line is temporarily stored in a queue of a memory controller, the CLWB instruction assumes that the cache line is automatically and safely written to PM by a power-fail feature called asynchronous DRAM refresh (ADR). Moreover, the order of writes to PM must be preserved using memory fence operations such as *sfence*.

In addition to data persistence, it is necessary to guarantee transaction atomicity on PM. As data becomes persistent when written back to PM, we need a mechanism to roll-back writes to PM when a transaction aborts before a commit. We implement atomic transactions on PM by modifying the source code of Tarantool with the *libpmemobj* library of the NVM Library [18] that provides APIs for transactional operations. The operations in a block between two special macros (TX_BEGIN and TX_END) are processed transactionally. In this block, a memory region to be modified can be specified with the *pmemobj_tx_add_range_direct* function. It takes a snapshot of the region and saves it in an undo log. If a transaction aborts, all the changes within the range are rolled-back automatically. Moreover, the memory region is automatically persisted when committing the transaction with the *pmemobj_persist* function, which internally calls CLWB and sfence instructions. Since we select the tree data structure for index and data in the database is maintained in the leafs, we guarantee the persistence of them and the atomicity of functions that updates them using these APIs.

4.3 Trade-Off on Hybrid Main Memory

When an application which guarantees data persistence with logging runs on a platform that apples hybrid main memory with storages, a new trade-off must be taken into account to maximize the overall performance. Table 3 summarizes the pros and cons of DRAM and PM in this situation. If data is placed on DRAM, it can be accessed faster because the access speed of DRAM is higher than that of PM. However, the persistence of the data must be guaranteed with logging. The overhead of logging is composed of (1) CPU cycles to construct a log entry for each data update and (2) the I/O time to completely write the entry to storages [13]. Since this overhead is not trivial, it may significantly hurt the overall performance. On the other hand, if the data is placed on PM, logging is unnecessary because data persistence is guaranteed with the instructions described in the previous section. However, the slower accesses (especially writes) to PM and the overhead of the additional instructions on PM may hurt the overall performance.

We expect that the overhead of logging data on DRAM becomes larger than the overhead of placing the data on PM instead of DRAM if the data is written frequently. The impact of slower writes to PM on the overall performance can be hidden by exploiting instruction- and memory-level parallelism on modern computer systems. However, the logging overhead cannot be hidden because transactions must be processed atomically, which implies that the following transaction cannot be processed until the log entry of the current transaction is completely written to a storage on a commit. We quantitatively evaluate this trade-off in the next section.

5 EVALUATION

In this section, we evaluate the trade-off between placing data on faster DRAM with logging and slower PM without logging, by running Tarantool with a micro-benchmark on our PM emulation platform. We first explain the implementation of the benchmark and then show the evaluation results.

5.1 Micro-benchmark

We implement a micro-benchmark that randomly executes select (i.e., read) and replace (i.e., write) operations to records in the database of Tarantool. This micro-benchmark imitates the implementation of the Yahoo! Cloud Serving Benchmark (YCSB) [4] that is commonly used to evaluate NoSQL databases. We use our own benchmark instead of YCSB, because YCSB simulates communications between clients and databases but we aim to evaluate the performance of the database itself.

The pseudo code of our micro-benchmark is shown in Algorithm 1. The inputs are the number of records stored in the database, the number of total operations, and the ratio of replace operations to the total operations (write ratio). Each record contains a key and ten fields, each of which corresponds to 100 B data. Thus, the total data size of each record is 1000 B. This benchmark first inserts all of the records to the database and then executes the specified number of operations. For each operation, it randomly chooses either select or replace operation and executes the selected one to a randomly selected record. The numbers of select and replace operations (#reads and #writes) are calculated based on the total number of operations and the write ratio, respectively. If the number of one operation executed reaches the one's number, that operation is not selected after that. In our experiments, we set the numbers of records and total operations to one million and change the write ratio from 0% to 100% at the step of 10% (e.g., 0.9 million select operations and 0.1 million replace operations are totally executed if the write ratio is 10%). The graphs in the next section report the throughput (operations per second) of Tarantool during the one million operations as a performance metric. Note that no snapshot is taken in Tarantool during the execution of this benchmark because the execution finishes tens of seconds at the most.

Algorithm 1 Pseudo code of our micro-benchmark

```
Input: #records, #total_operations, write_ratio
Output: operations_per_sec
 1: #reads = #total_operations * (100 - write_ratio) / 100
 2: #writes = #total_operations * write_ratio / 100
 3: record ← create_1000B_record()
 4: for $i$ = 1 to #records do                 # Insert all records at first
 5:     $insert$($i$, record)
 6: end for
 7: read_cnt ← 0
 8: write_cnt ← 0
 9: new_record ← create_1000B_record()
10: t1 ← get_current_time()
11: for $i$ = 1 to #total_operations do
12:     if read_cnt > #reads then
13:         op ← 2                              # Choose $replace$
14:     else if write_cnt > #writes then
15:         op ← 1                              # Choose $select$
16:     else
17:         op ← random value (1 to 2)          # Randomly choose $select$ or $replace$
18:     end if
19:     key ← random value (1 to #records)
20:     if op == 1 then
21:         $select$(key)
22:         read_cnt ← read_cnt + 1
23:     else
24:         $replace$(key, new_record)
25:         write_cnt ← write_cnt + 1
26:     end if
27: end for
28: t2 ← get_current_time()
29: operation_per_sec ← #total_operations / (t2 - t1)
```

Table 4: Description of four types of executions.

Label	Data placement	Logging
DRAM w/o logging	DRAM	–
WAL-fsync	DRAM	WAL
WAL-nowrite	DRAM	WAL without writes to log
PM	PM	–

5.2 Methodology

In order to evaluate the overheads of logging data on DRAM and placing data on PM, we compare four types of executions summarized in Table 4: *DRAM w/o logging*, *WAL-fsync*, *WAL-nowrite*, and *PM*. DRAM w/o logging is an execution that places all data on DRAM without logging, which does not incur the both overheads. WAL-fsync places all data on DRAM and guarantees data persistence with the write-ahead logging (WAL). WAL-nowrite is a similar execution to WAL-fsync but it eliminates the I/O time from the logging overhead by skipping writes to the log file on the storage. Although WAL-nowrite assumes the techniques to write log entries to PM, as proposed by Fang et al. [6] and Gao et al. [8], it does not include the impact of the PM latency. Finally, PM is an execution that places all data on PM and guarantees transaction atomicity and data persistence with the libpmemobj library. As we change the PM latency to 2x, 4x, and 8x of the DRAM latency, the value is added as the suffix to PM (e.g., *PM-4x* corresponds to the execution on PM with the 4x latency).

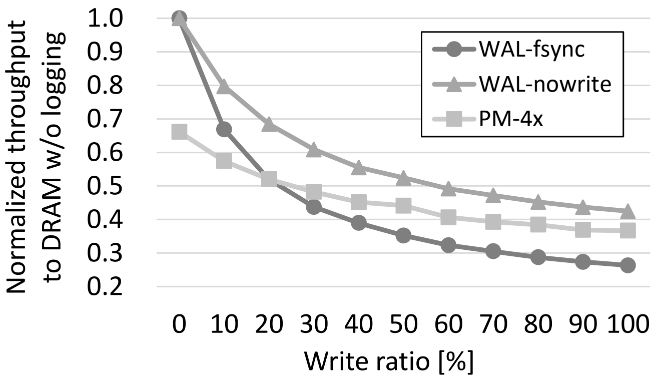

Figure 2: Normalized throughput to DRAM w/o logging. The x-axis indicates the ratio of writes (i.e. replace operations) to 1,000,000 operations.

5.3 Results

Figure 2 plots the throughput (operations per second) of Tarantool along with the write ratio of our micro-benchmark. The PM latency is set to 4x of the DRAM latency, and the throughput is normalized by that of DRAM w/o logging. At first, we can know the logging overhead from the result of WAL-fsync. It shows that the logging overhead becomes larger as the write ratio is increased. This is intuitive because the number of writes at a certain epoch increases as the write ratio is increased. If the write ratio is 50% (i.e., the workload has 50% reads and 50% writes), the throughput of Tarantool degrades to 35%.

On the other hand, the overhead of placing data on PM instead of DRAM is obtained from the result of PM-4x. We can see that the throughput of PM-4x is much lower than that of WAL-fsync when the write ratio is 0% and 10%. In these cases, the 4x higher read latency of PM largely affects the throughput. However, the throughput of PM-4x is expected to be better on real PM because its actual read latency is close to the DRAM latency. In contrast, PM-4x outperforms WAL-fsync at the over 20% write ratio. This result means that the logging overhead becomes larger than the overhead of placing data on PM if the data is frequently written, which verifies our expectation mentioned in Section 4.3. That is, it is better for performance to place frequently written data on PM instead of DRAM, when an application which requires logging is executed on hybrid main memory.

Finally, the result of WAL-nowrite indicates the overhead of creating the log entries. This overhead remains even if WAL is optimized to write the entires to PM instead of a storage. The figure shows that this overhead is not trivial, but WAL-nowrite outperforms PM-4x. This means that the overhead of creating log entries is smaller than that of placing data on PM. However, WAL-nowrite ignores the overhead of writing the log entries to PM. With this overhead, the throughput of WAL-nowrite will be worse and become compatible to that of PM-4x. In this case, placing data on PM can achieve a comparable throughput to a PM-based logging while saving the capacity of log entries on PM.

Figure 3 plots the throughput of PM with various latencies. Note that the throughput is normalized by that of WAL-fsync, not DRAM

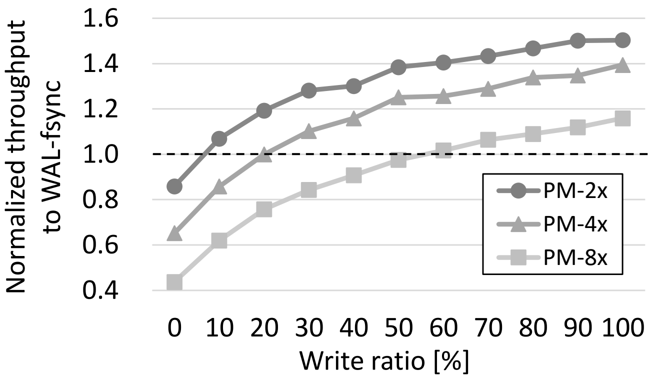

Figure 3: Normalized throughput of PM with various latencies to WAL-fsync.

w/o logging. We can observe that the throughput of PM is largely affected by the PM latency. If it becomes 2x of the DRAM latency, we can significantly improve the throughput by placing data on PM compared to placing the data on DRAM with logging (e.g., 38% at the 50% write ratio). Even if the PM latency is 8x longer than the DRAM latency, heavily written data (over 60% write ratio) should be placed on PM to improve the throughput.

6 CONCLUSIONS

In this paper, we focus on applications which guarantee data persistent with logging and suggest a new trade-off between DRAM and PM for data placement on platforms that apply hybrid main memory with storages. Placing data on DRAM can avoid the slower accesses to PM, but it incurs the logging overhead. On the other hand, placing data on PM can avoid the logging overhead while it suffers from slower accesses to PM. Our quantitative evaluation using a PM emulation platform reveals that, compared to placing frequently written data on DRAM with logging, the performance of an in-memory database can be improved significantly by placing the data on PM. This is because the logging overhead becomes larger than the overhead of placing data on PM when the data is heavily written. Based on these results, we need to rethink data placement on hybrid main memory for applications which guarantee data persistence. As future work, we will implement a logging technique to write log entries to PM and quantitatively compare it with placing data on PM.

ACKNOWLEDGMENTS

We would like to thank Dieter Kasper and Andreas Blümle for helping with the persistent memory emulation platform.

REFERENCES

[1] Katelin A Bailey, Peter Hornyack, Luis Ceze, Steven D Gribble, and Henry M Levy. 2013. Exploring Storage Class Memory with Key Value Stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW '13)*. ACM, 4:1–4:8.

[2] Shimin Chen, Phillip B Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. In *Proceedings of 2011 fifth Biennial Conference on Innovative Data Systems Research.*

[3] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, 133–146. http://doi.acm.org/10.1145/1629575.1629589

[4] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, 143–154.

[5] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, 15:1–15:16.

[6] Ru Fang, Hui-I Hsiao, Bin He, C Mohan, and Yun Wang. 2011. High Performance Database Logging using Storage Class Memory. In *Proceeding of 2011 IEEE 27th International Conference on Data Engineering (ICDE'11)*. 1221–1231.

[7] Mike Ferron-Jonesi. 2017. A New Breakthrough in Persistent Memory Gets Its First Public Demo. https://itpeernetwork.intel.com/new-breakthrough-persistent-memory-first-public-demo/. (May 2017). Last access: August 2017.

[8] Shen Gao, Jianliang Xu, Theo Härder, Bingsheng He, Byron Choi, and Haibo Hu. 2015. PCMLogging: Optimizing Transaction Logging and Recovery Performance with PCM. *IEEE Transactions on Knowledge and Data Engineering* 27, 12 (2015), 3332–3346.

[9] Thai Le. 2016. How to Emulate Persistent Memory on an Intel® Architecture Server. https://software.intel.com/en-us/articles/how-to-emulate-persistent-memory-on-an-intel-architecture-server. (September 2016). Last access: August 2017.

[10] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (jan 2010), 143.

[11] Soyoon Lee, Hyokyung Bahn, and Sam H Noh. 2014. CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures. *IEEE Trans. Comput.* 63, 9 (2014), 2187–2200.

[12] Dong Li, Jeffrey S Vetter, Gabriel Marin, Collin McCurdy, Cristian Cira, Zhuo Liu, and Weikuan Yu. 2012. Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. IEEE Computer Society, 945–956.

[13] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking Main Memory OLTP Recovery. In *Proceeding of 2014 IEEE 30th International Conference on Data Engineering (ICDE '14)*. 604–615.

[14] Jeffrey C Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. 2009. Operating System Support for NVM+DRAM Hybrid Main Memory. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS'09)*. USENIX Association, 14.

[15] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162.

[16] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. 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)*. ACM, 8:1–8:7.

[17] Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, and Peter Bumbulis. 2015. Instant Recovery for Main Memory Databases. In *Proceedings of 2015 Seventh Biennial Conference on Innovative Data Systems Research (CIDR'15).*

[18] pmem.io. 2017. Persistent Memory Programming. http://pmem.io/. (2017). Last access: August 2017.

[19] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, 85–95.

[20] Andy M Rudoff. 2016. Deprecating the PCOMMIT Instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction. (September 2016). Last access: August 2017.

[21] Tarantool. 2017. Get your data in RAM. Get compute close to data. Enjoy the Performance. https://tarantool.org/. (2017). Last access: August 2017.

[22] Yiying Zhang and Steven Swanson. 2015. A Study of Application Performance with Non-Volatile Main Memory. In *Proceeding of the 2015 Symposium on Mass Storage Systems and Technologies (MSST'15)*. 1–10.