2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)

Fine-Grained Data Committing for Persistent Memory

Tianyue Lu, Yuhang Liu, Mingyu Chen
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
{lutianyue, liuyuhang, cmy}@ict.ac.cn

***Abstract*—Non-Volatile Memory (NVM) is better than traditional DRAM with respect to energy efficiency and larger capacity, so NVM has begun to be used as main memory. NVM provides data persistence that data written into NVM will not be lost during unexpected system failure occurs. Data persistence is mandatory for programs such as file system and database. However, traditional memory protocol cannot provide an mechanism for programs to guarantee data persistence because the write instructions do not ensure that data would be eventually written into the memory media. Furthermore, extra global operations such as PCOMMIT for data committing could incur significant performance loss, especially for multi-task programs. To address this issue, we propose a hardware-software coordinated mechanism to achieve low-overhead data committing. Write queues in memory controller are divided into multiple sub-queues for monitoring write commands for different address ranges. Programs can query write queues to check the execution status of previous written commands through a series of OS-managed library APIs. Fine-grained data committing can reduce the interferences among processes effectively. Extensive evaluations show that per-task data committing brings an average 1.78x performance improvement than original global committing mechanism and accelerates the data committing by 2.07 times.**

***Index Terms*—Non-Volatile Memory, Persistent Memory, Data Committing**

I. Introduction

Big data era has made new on-line transaction processing (OLTP) applications supporting increasingly more concurrent users and data processing [1]–[4]. Database management systems (DBMSs) are critical applications among OLTP applications which are responsible for ensuring data persistence. Data persistence means that updated data is written on non-volatile devices thus prevents data loss after a power corruption.

As a traditional method to measure data persistence, DBMS writes data to non-volatile storages periodically, like a SSD or HDD. We can call it a synchronization point, when DBMS program writes all its data which need persistence into non-volatile storage. To ensure data persistence, program will block all new data writes during a synchronization period. When synchronization period finishes, all data written before it are committed to non-volatile storage and program is permitted to issue new data writes. Data in non-volatile memory exist even after power corruption and programs can restore their data after system restarting.

This work is supported by National Key Research and Development Plan of China 2017YFB1001600, NSFC (National Science Foundation of China) No. 61331008 ,61521092 and 61772497, State Key Laboratory of Computer Architecture foundation under grant CARCH2601 and Huawei YBCB2011030. Corresponding author is Yuhang Liu.

TABLE I
PERFORMANCE OF DIFFERENT NON-VOLATILE DEVICES

	Write latency	Access granularity
NVM (PCM)	hundreds of nanoseconds	64B
SSD	hundreds of microseconds	4KB
Disk	tens of milliseconds	4KB

Non-Volatile Memory (NVM) has been emerged as a new main memory system, like PCM or RRAM [5]–[11]. Compared to conventional DRAM main memory, NVM has higher density and lower energy. Besides, NVM provides data persistence which means data in NVM will not be loss after a power fail [12]–[16]. Thus, NVM is able to replace the SSD or HDD as the non-volatile device to achieve data persistence in DBMS applications. Table I shows the performance parameters of different non-volatile devices. In DBMS, one synchronization period is needed each time data is modified. So the length of synchronization period has a great influence on system performance. Especially in multi-thread OLTP applications, data update could be very frequent.

Although NVM provides a physical feature of data persistence, current memory hierarchy does not support writing data into main memory directly. This is due to complex memory hierarchies including CPU caches [17], [18]. When application issues memory write instructions, these instructions are committed after data are written into L1 CPU cache, and all cache replacement and eviction are done by hardware in the background.

A feasible way to achieve data committing on memory is using assembly instructions CLFLUSH and MFENCE. But in complex memory hierarchy, CLFLUSH cannot ensure that data will not be buffered in memory controller or other buffers between CPU cache and memory chips. As a solution, PCOMMIT [19] has been proposed in Intel ISA to flush all data from memory controller to memory chips. But PCOMMIT will barrier all write commands issuing from all CPU cores. We can call this mechanism global committing and it succeeds providing a way to commit data to persistence, but it takes coarse granularity and affects system performance.

As an improved method, Intel has deprecated PCOMMIT and presents Asynchronous DRAM refresh (ADR) technology [20]. ADR technology can ensure that, all written data buffered in memory controller is able to flushed into memory when a power loss happens. Using ADR, PCOMMIT is not needed because when data is flushed from cache into memory controller, its data persistence can be ensured by ADR mechanism. However, ADR is not supported by all CPU. Especially, NVM-based main memory is probably configured as DRAM-NVM hybrid memory and DRAM is used as L4 cache. In hybrid memory case, ADR only flush data from memory controller to off-chip DRAM cache which is not a non-volatile device. This means that, ADR technology is not able to provide data persistence in complex DRAM-NVM hybrid memory situation.

In this paper, we propose a fine-grained memory committing mechanism. In a hardware-software coordinated way, memory committing only blocks write commands of partial address space, especially does not affect other parallel-running threads. Write queue in memory controller is separated into multiple queues and each of them corresponds to a memory address range. These ranges of address are defined by applications using a specific API. Memory commands of different address ranges are buffered in different corresponding queues. In memory committing period, write commands in different queues are committed independently and will not affect commands in other queues.

Our contribution of this paper is that, we propose an effective persistence mechanism based on NVMM committing. This mechanism uses separate memory write queues in memory controller to achieve fine-grained memory committing. The fine-grained method reduces the time overhead of memory committing process and avoids all running threads to be paused in a committing period. The evaluations show that, our fine-grained mechanism has a 1.78 times performance improvement over traditional global committing mechanism.

The rest of paper is organized as: Section 2 introduces the backgrounds of NVM and data committing. Section 3 introduces our proposed scheme. Section 4 shows evaluations of different memory committing mechanisms. Finally, Section 5 concludes the whole paper.

II. Backgrounds

In this section, we will introduce the background of data persistence and data committing mechanism on memory interface.

A. Data Persistence and NVM

DBMS applications constitute an important type of big data applications. It provides an on-line data processing servicing parallel users and threads. Furthermore, it demands data persistence that data of database must not be lost. In modern computer system, many volatile devices are used to improve data access speed, like DRAM memory and SRAM cache. During normal program running, updated data are only written into volatile devices. To achieve data persistence, data of database must be written back to non-volatile devices. And Non-Volatile Memory (NVM) provides a potential mechanism that data in NVM is persistent without writing back to disk.

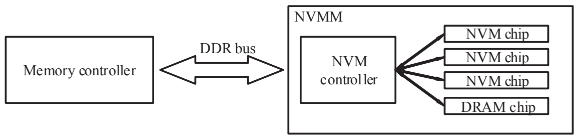

Fig. 1. Memory interface of NVM and NVM controller

NVMs such as PCM are byte-addressable persistent devices expected to be 100x faster (read-write performance) compared to current SSDs. Also, NVM has a 2x-4x higher density than DRAM because it can store multiple bits per cell, which makes NVM a suitable candidate for replacing DRAM as high-capacity memory. Non-volatile main memory (NVMM) architecture is shown in Figure 1. NVM's read latency is comparable to DRAM latency, but the write latency is 5x-10x slower. NVMM uses DDR-like memory protocol and NVM controller is added at NVM's side. NVM controller manages all memory commands transferred from memory controller. Considering the architecture of NVMM could be complex like hybrid DRAM-NVM memory, NVM controller is necessary to provide an unified interface to CPU side.

B. Memory Committing

In DBMS applications, data committing happens frequently that most committing period could only have one or a few writes. This makes data committing take a large proportion of program running time. Usual data committing period is shown in Figure 2a. Compared to traditional mechanism which write data back to disk, writing data back to memory could bring a great performance improvement.

Different with disk I/O operations, memory write instruction will be finished right after data are written on cache instead of main memory. Only when data is evicted from cache, a main memory write command is issued. But data eviction is controlled by hardware in the background. So if we use non-volatile memory to replace non-volatile disk as the data persistent devices, programs cannot simply use memory write instructions to replace disk write instructions.

A feasible solution is using CLFLUSH and PCOMMIT in data committing as shown in Figure 2b. CLFLUSH flushes appointed data out of cache. In this case, flushed data is buffered in memory controller and forms a memory command in write queue. PCOMMIT flushes all memory write commands out of memory controller which means all corresponding data are written into main memory. But PCOMMIT acts on memory controller and it blocks all memory write commands from issuing into memory controller. This means that, all other running threads must stop data committing before current committing period is finished. As data committing could be very frequent in DBMS applications, this method could takes a serious time overhead of memory committing.

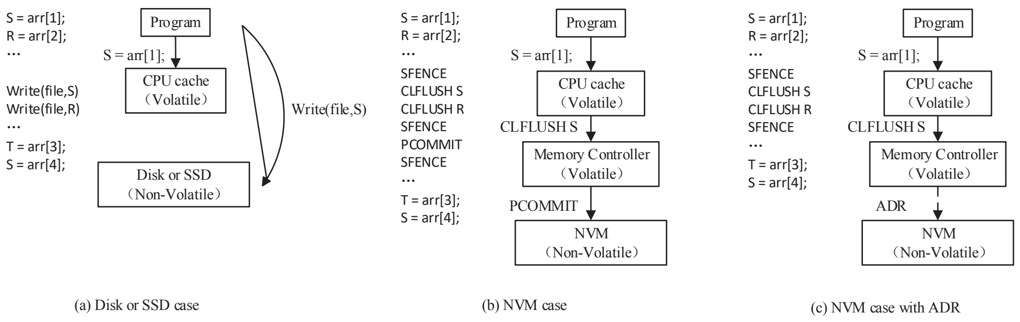

Fig. 2. Data committing of different non-volatile devices

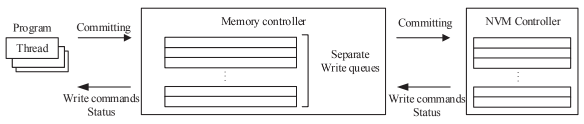

Fig. 3. Design overview of Fine-grained Memory Committing

Another potential technology is ADR. ADR technology can ensure that, all written data buffered in memory controller is able to flushed into memory when a power loss happens. This is achieved by that a battery is configured in memory controller and it will satisfy the power of data flushing from memory controller to main memory. Using ADR, data in memory controller is regarded as persistent so that PCOMMIT is not needed in memory committing period as shown in Figure 2c. But ADR is not supported by all types of CPU and large battery is not easy to achieve. Also, ADR is not suitable to which NVM is combined with DRAM cache as hybrid memory. Because DRAM cache is a large non-volatile L4 cache, battery is hardly large enough to flush all data in DRAM cache into NVM.

III. Fine-Grained Data Committing

Based on our analysis and current works, we propose fine-grained hardware-software coordinated memory committing. In Section III-A, we will introduce our design ideas and in Section III-B and III-C, hardware modification and software library functions are detailed.

A. Design

The overview of fine-grained memory committing is shown in Figure 3. Our first point is that write commands are buffered in separate write queues in memory controller Thus at the time of memory committing, only one write queue is needed to be blocked. As each write queue corresponds different memory space, only write commands of relative memory addresses are blocked. The binding of memory spaces and write queues are done by OS-managed APIs so that threads are able to use different write queues to avoid interfere during memory committing period.

Our second point is that memory controller communicates with NVM controller actively so that status of write commands

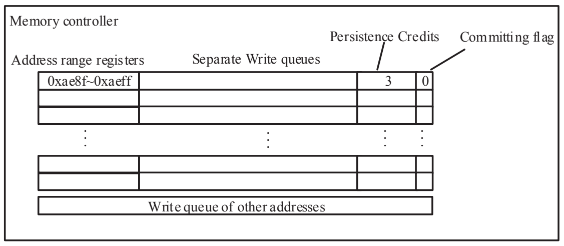

Fig. 4. The structure of memory controller with separate write queues

in NVM controller can be tracked by memory controller. Programs can also use specific address to check status recorded in memory controller. Committing operations are started by programs and managed by memory controller which flushes write queues to NVM controllers and track the statuses.

B. Implementation:Hardware

As stated in Section II, CPU cores cannot track the execution state of write instructions, so we first separate write queue in memory controller into several ones. The structure of new write queues is shown in Figure 4. Write queues receives all write commands issued from programs to memory controller. And in queues, write instructions with specific addresses are buffered in separated queues according to a registered address space list. Each queue corresponds to a memory space and the address ranges are appointed in address range registers by OS. Write queues also keep persistence credit counters for each queues to identify the execution state of write commands. Every write commands occupies one persistence credit of corresponding queue. When the credit value equals to its maximum value, it identifies that all previous write instructions are completed. Programs can query these credits to check the statuses. Write commands of addresses other than registered addresses will be buffered in another individual queue on memory controller.

The persistence credit value is synchronized with media controller of NVM. As memory module doesn't have an active mechanism to announce the credit value update to memory controller in standard interface, the credit update is queried by memory controller with a read command or returned with

```
int PERSISTENCEMALLOC(int size)
bool PERSISTENCECOMMIT(int QueueNum)
void PERSISTENCERELEASE(int BasicAddr)
```

Fig. 5. Library functions for per-task committing

other read commands coded with returned data. For software to access these credits, read commands to special addresses are persistence query accesses. As response to these persistence query accesses, memory controller will return the credit value of corresponding queue and no longer transfer it to memory chip as common read commands.

To support persistence credits, NVM controller should also add a set of persistence credit registers and address range registers. The extended NVM controller is similar to memory controller with additional address range registers and persistence credit registers. In address range registers, NVM controller keeps the same value with memory controller that their values are synchronized immediately after they are set in memory controller. As address ranges in memory controller are written by specific write commands, those in NVM controller are also written by same value. Persistence credit registers are updated by NVM controller itself. When a write command is issued, persistence credit decreases by 1. And after it is finished, credit increases by 1. Newest persistence credits change is synchronized back to memory controller. The synchronization operations are issued by memory controller that, when memory controller finds that a credit value is low, it sends a specific read command. This read command to specific address will be captured by NVM controller and the credit update information will be encoded in returned data. The interval of that memory controller requires for persistence credit update could be dynamic. The more often it requires, the more read commands are sent on memory bus. On the contrary, new write commands could have to wait for credit because the credit value is 0 in memory controller and the update information has not arrived.

C. Implementation:Software

In this section, we will introduce the software management of fine-grained committing. Figure 5 shows all OS-managed library APIs of fine-grained committing. Function PersistenceMalloc is used for programs to allocate a memory space and bind it to a queue in write queue which shown in Figure 6. The return value of PersistenceMalloc is the queue number of write queues. When programs want to confirm the data persistence, function PersistenceCommit would be executed and data persistence is confirmed by credits. When programs demand no data persistence, function PersistenceRelease can unbind and release the memory space on write queues.

When program wants to monitor data persistence on a memory space, it can use PersistenceMalloc to allocate a memory space and write its address range into address range register. PersistenceMalloc is an OS-managed function like Malloc that OS allocates a memory space and selects a write queue as the register target. And then, the basic address and space size is assigned into corresponding address range register. At last, the queue number is return to program. This number will be used in data committing period. In fact, multiple address ranges can be bound to one queue. And OS can monitor the usage of write queues to select the queue with the lowest utilization for new binding address range. PersistenceRelease is reverse operations of PersistenceMalloc that it releases the assigned memory space.

```
int PERSISTENCEMALLOC(int size)
{
   BasicAddr = MALLOC(size);             //Use function malloc to alloc a normal address
                                            space
   Select QueueNum;                      //OS select a queue of write queues in memory
                                          controller
   AddrStart[QueueNum] = BasicAddr;
   AddrSize[QueueNum] = ArraySize;       //Register the start address and space size in
                                          address range register
   return QueueNum;                      //Return queue number to program
}
```

Fig. 6. Pseudo code of function PersistenceMalloc

```
bool PERSISTENCECOMMIT(int QueueNum)
{
     SETCOMMITFLAG(QueueNum);                           //Set committing flag of corresponding
                                                          write queue
     LOCK(QueueNum);                                    //Block other write command to this queue
     while(PersistenceCredit[QueueNum] < max); //Read persistence credit value repeatedly,
                                                  until it returns to max value
     CANCELCOMMITFLAG(QueueNum);                        //End data committing period
     return 0;
}
```

Fig. 7. Pseudo code of function PersistentCommit

After allocating a special memory space for data persistence, all write commands to this space are tracked by corresponding write queue so that programs could use the normal write instructions to do write operations. Function PersistentCommit is used for programs to confirm that all previous written data have been written to persistent memory. Its pseudo code is shown in Figure 7. When a data committing operation starts, PersistentCommit will set a committing flag to indicate that memory controller will reject all other write commands to access this queue and send all buffered commands to memory chip, otherwise the persistence credit will be taken again by new write instructions which means that the credit value will never reach maximum. At the same time, access to the same queue from other threads should also be blocked. To multi-thread programs, the queue will locked before PersistentCommit is finished. Only when the persistence credit value reaches maximum and the corresponding queue is empty, all previous written data is assured to be persistent. After set committing flag, the value of persistence credit register of corresponding queue is queried repeatedly, until the credit value is the same with the maximum credit limit, which means that all previous

TABLE II
SYSTEM CONFIGURATIONS

ROB	2.7GHz, 256-entry, max fetch/retire per cycle: 4/2
L1 Cache	Private, 32KB, 4-way, 64B-block, 4-cycle hit
L2 Cache	Private, 256KB, 8-way, 64B-block, 10-cycle hit
L3 Cache	Shared, 1MB/core, 16-way, 64B-block, 40-cycle hit
DRAM Parameters	
Memory	2 64-bit Channels, 2 Ranks/Channel, 8 devices/Rank, 8 sub-ranks/rank, x8-width (1 device) sub-rank
PCM Parameters	
Timing	Read latency: 55ns, Write latency: 150ns
NVM controller	Read queues: 64 entries Write queue: 64 entries

TABLE III
WORKLOADS AND CONFIGURATIONS

Benchmark	Configuration
Echo	4 clients, 1 million transactions
YCSB	4 clients, 8 million transactions
TPC-C	4 clients, 1 million transactions
Redis	lru-test, 1 million keys
C-tree	4 clients, 100K INSERT transactions
Hashmap	4 clients, 100K INSERT transactions
Vacation	4 clients, 2 million transactions, 16 million tuples
Memcached	memslap, 4 clients, 100K operations
NFS	filebench, 8 clients
Exim	postal, 8 clients
MySQL	OLTP-complex, 4 clients

write commands are all transferred into memory chips. At this time, corresponding write queue is unlocked and new write operations to corresponding queue are allowed to be issued. Program will also receive the updated persistence credit and finish PersistentCommit function.

IV. EVALUATIONS

A. Experimental methodology

We use a cycle-accurate simulator DRAMSim2 [21] driven by Pin-generated trace for our evaluation [22]. The simulator is extended to provide Re-order Buffer (ROB) module to manage trace input and instruction pipeline. PCM timing is added in DRAMSim2. Specifically, we modify the original DRAM parameter to PCM parameter shown in Table II. Separate write queues and NVM controller is also added in DRAMSim2. To simulate persistence credit, we also add the credit mechanism that each write commands with specific memory address will take one credit, and memory controller will query updated credit using read command when a credit value is 0. The number of address range registers are 4 to each queue which means that one queue is able to correspond to 4 address spaces and each write queue has 8 queue entries. In experiment, we run a same CPU cycles to global committing and fine-grained committing and compare their performances.

We modify the original programs to add instructions for confirming persistence and credit synchronization operations are generated in simulator in need automatically. ROB is deployed one for each core and we run our benchmark programs in 16-core configuration. In workload, we use a persistent memory (PM) benchmark suite called WHISPER as the target programs [23]. WHISPER includes 11 programs shown in Table III.

B. Performance results

Figure 8 shows the performance comparison of programs using different persistence committing mechanisms. In all the programs, the average ratio of read and write is 1.96:1. Fine-grained committing shows an improvement of 1.78 times over global committing due to that interference of threads is low. In Figure 9, different execution times of committing operations also shows that programs runs faster in fine-grained commit because the time cost of single committing operation is reduced by 60.4%.

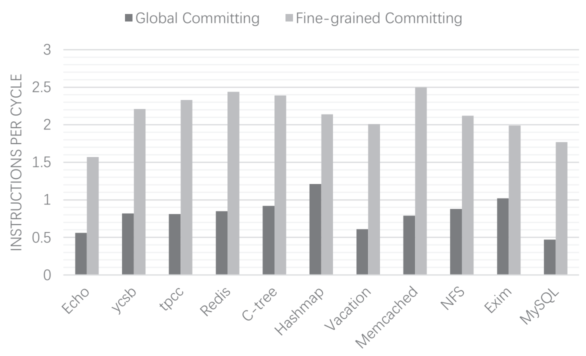

Fig. 8. Instructions Per Cycle (IPC) of Global and Fine-grained committing

Figure 10 shows the performance results of different data committing frequency. If we reduce data committing frequency that starts a data committing period after multiple data updates, program will be less interrupted by committing periods. In Figure 10, we find that in the case of data committing is issued every 100 data updates, fine-grained committing has a performance improvement of 6.0% but if data committing is issued every update, the improvement may reach 100%.

We also simulate the credit synchronization overhead. In fine-grained committing, each queue has its own persistence credit. Global committing is considered as using one synchro-

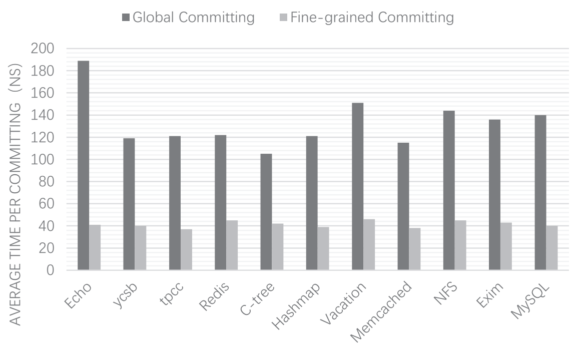

Fig. 9. Time spending on single data committing period of Global and Fine-grained committing

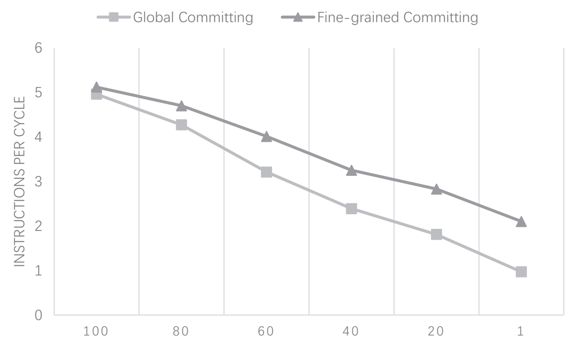

Fig. 10. Average Instructions Per Cycle (IPC) of Global and Fine-grained committing in different data committing frequency

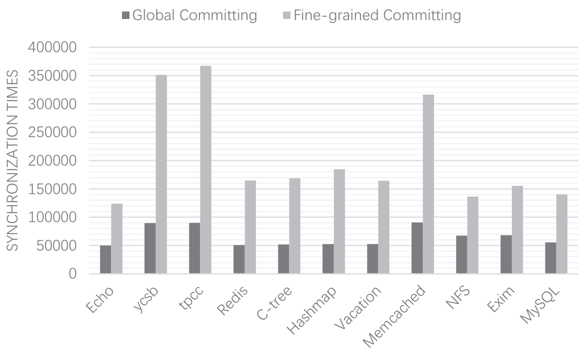

Fig. 11. Synchronization times of persistence credits of Global and Fine-grained committing

nization operation in one committing period. More credits take more synchronization overhead. Figure 11 shows the extra synchronization operations of different method. We can see that fine-grained mechanism takes synchronization operations 2.4 times more than global committing on average.

V. Conclusions

Non-Volatile Memory is expected to become the mainstream memory, with large capacity and low power consumption. Also, new features of NVM are also worth exploring, especially the data persistence. But in traditional memory protocol, data persistence is not supported that programs are hard to efficiently confirm persistence. Our proposed fine-grained memory committing mechanism provides a hardware-software coordinated way for programs to acknowledge the status of previous write commands. With this mechanism, write commands with registered addresses are monitored by separate write queue and programs can use read instructions to read the persistence credits. The time spending on data committing operation is shorter because the data committing period will not affect other commands issued by other threads. Evaluations shows that, fine-grained committing improves the system performance by 1.78 times compared with global committing.

References

[1] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–499, IEEE, 2014.

[2] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi, "Database servers on chip multiprocessors: Limitations and opportunities," in *Proceedings of the Biennial Conference on Innovative Data Systems Research*, no. DIAS-CONF-2007-008, 2007.

[3] "Tokyo Cabinet: a modern implementation of DBM." http://fallabs.com/tokyocabinet/, 2014.

[4] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: reducing consistency cost for nvm-based single level systems," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pp. 167–181, 2015.

[5] S. Chhabra and Y. Solihin, "i-NVMM: a secure non-volatile main memory system with incremental encryption," in *ISCA 2011*.

[6] M. Qureshi, V. Srinivasan, and A. Rivers, "Scalable high performance main memory system using phase change memory technology," in *ACM SIGARCH Computer Architecture News*, 2009.

[7] Micron, "NVDIMM." http://www.micron.com/products/dram-modules/nvdimm.

[8] SanDisk, "ULLtraDIMM SSDs." http://www.sandisk.com /enterprise/ulltradimm-ssd/.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pp. 133–146.

[10] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[11] M. Wu and W. Zwaenepoel, "envy: a non-volatile, main memory storage system," in *ACM SIGOPS Operating Systems Review*, 1994.

[12] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, "Software persistent memory," in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC)*, pp. 319–331.

[13] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 14–23, 2009.

[14] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 91–104.

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

[16] Y. Lu, J. Shu, and L. Sun, "Blurred Persistence: Efficient Transactions in Persistent Memory," in *ACM Transactions on Storage (TOS) - Special Issue on Massive Storage Systems and Technologies (MSST 2015)*, 2016.

[17] Y. Liu and X.-H. Sun, "Lpm: Concurrency-driven layered performance matching," in *International Conference on Parallel Processing*, pp. 879–888, 2015.

[18] Y. H. Liu, "c^2-bound: a capacity and concurrency driven analytical model for many-core design," in *the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2015.

[19] Intel, "Intel Architecture Instruction Set Extensions Programming Reference," 2014.

[20] D.Williams, "Replace pcommit with ADR or directed flushing,"

[21] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," jan.-june 2011.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI 2005*, pp. 190–200.

[23] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *ASPLOS'17 Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 135–148, 2017.