Quick-and-Dirty: Improving Performance of MLC PCM by Using Temporary Short Writes

Mingzhe Zhang*†, Lunkai Zhang§, Lei Jiang¶, Frederic T. Chong§ and Zhiyong Liu*‡
*Beijing Key Laboratory of Mobile Computing and Pervasive Device, ICT, CAS, Beijing, China
†University of Chinese Academy of Sciences, Beijing, China
‡State Key Laboratory of Computer Architecture, ICT, CAS, Beijing, China
§Computer Science Department, University of Chicago
¶Department fo Intelligent Systems Engineering, Indiana University Bloomington
{zhangmingzhe, zyliu}@ict.ac.cn, lunkai@uchicago.edu, jiang60@iu.edu, chong@cs.uchicago.edu

***Abstract*—Low write performance is a major obstacle to the commercialization of MLC PCM. One opportunity for improving the latency of MLC PCM writes is to use fewer SET iterations in a single write. Unfortunately, the data written by these short writes have significantly shorter retention time and thus need frequent refreshes. As a result, it is impractical to use these short-latency, short-retention writes globally. In this paper, we analyze the temporal behavior of write operations in typical applications and propose *Quick-and-Dirty (QnD)*, a lightweight scheme to improve the performance of MLC PCM. *QnD* dynamically performs the short-latency, short-retention write when write operations are bursty, and then uses short-latency, short-retention writes to mitigate the short retention problem when memory system is relatively quiet. Our experimental results show that *QnD* improves performance by 30.9% on geometric mean while still providing acceptable memory lifetime (7.58 years on geometric mean). We also provide sensitivity studies of the aggressiveness, memory coverage and granularity of *QnD* technique.**

***Index Terms*—MLC PCM, performance, lifetime, retention time, tradeoff**

I. Introduction

Phase Change Memory (PCM) is considered to be a potential alternative technology to DRAM [1], [11], which provides high integration density, low power consumption, and has no volatility. Additionally, by carefully controlling the resistance, PCM allows the storage of multiple bits in a single cell (Multi-Level Cell PCM, MLC PCM) [9], [10]. To achieve fine-grained resistance control, however, MLC PCM requires an iterative write algorithm. This requirement significantly degrades the write performance of MLC PCM.

One way to speed up the write operation of MLC PCM is to include fewer SET iterations in a single write operation, but this comes at the cost of shorter data retention. For example, according to Table I, a 7-SETs-Write provides a retention time of 3054.9 seconds, while a 3-SETs-Write only achieves a retention time of 2.01 seconds. As a result, data written by these short latency writes need to be refreshed more frequently. This in turn hurts the memory lifetime and increases the system energy.

In this paper, we propose *Quick-and-Dirty (QnD)*, a simple but effective scheme to improve the write performance of MLC PCM. *QnD* uses short-latency, short-retention writes when write operations become the system performance bottleneck. It then uses long-latency, long-retention refresh to alleviate the retention problem left by the short writes. It does this later when the memory system is not busy. Compared to other techniques that use the *write latency vs. retention* trade-off in MLC PCM, *QnD* requires trivial overhead and does not require modification (software or hardware) outside the memory controller. Our experiment shows that, on geometric mean, *QnD* improves system performance 30.9% while achieving an affordable memory lifetime of 7.58 years.

This work is supported in part by NSFC grants No. 61520106005 and 61521092.

II. Background and Motivation

A. Background

As is shown in Figure 1, MLC PCM uses multiple resistance ranges to represent the multi-bit digital value in one cell. Because of narrow distribution of each resistance state, MLC PCM cells suffer from short retention time caused by resistance drift (i.e., the spontaneous increase of resistance in PCM cells). One way to combat such short retention time is to create large *guardbands*, which are resistance ranges deliberately left unused between two adjacent digital values. Though larger *guardbands* can deliver longer retention time, they require more accurate writes, which in turn need more SET operations and thus a longer write latency. As a result, there is a trade-off between write latency and retention time in MLC PCM.

To explore such a trade-off , we model a MLC PCM cell in detail with the same model used in [2] and the parameters from latest 20nm PCM chip [3]. In our model, a MLC PCM write has a RESET operation (100ns, 50μA) followed by several SET operations (150ns). Varying numbers of iterations (from 3 to 7) require different current amplitudes (from 42uA to 30uA). The model results are presented in Table I. We can see that a 7-SETs-write achieves a retention time of 3054.9 seconds, while a 3-SETs-write only 2.01 seconds retention. However, 7-SETs-write costs more than double the latency than 3-SETs-write (1150ns vs. 550ns). Note that, in our model we assume all write operations achieve the same endurance irrespective of the number of their SET iterations [4].

B. Motivation

Figure 2 shows the time pattern of write operations in our baseline system with 7-SETs-Write (described in Section

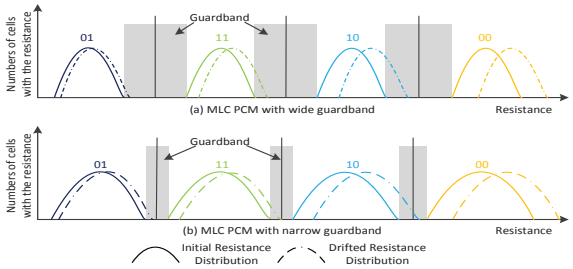

Fig. 1. Resistance drift problem in MLC PCM with guardband technique.

TABLE I
THE TRADE-OFF BETWEEN LATENCY AND RETENTION OF MLC PCM WRITE OPERATIONS.

Write Type	Latency (in ns.)	Retention (in seconds)	Current (in μA)	Norm. Energy
3-SETs-Write	550	2.01	42	0.84
4-SETs-Write	700	24.05	37	0.869
5-SETs-Write	850	104.4	35	0.972
6-SETs-Write	1000	991.4	32	0.975
7-SETs-Write	1150	3054.9	30	1

IV-A) running benchmark `libquantum`. We can see that, there are obviously *Busy Phases* (e.g., 0s to 0.37s), *Idle Phases*(e.g., 1.1s to 1.2) and *Relatively Idle Phases* (e.g., 1.23s to 1.35s) during the running time. Note that, these memory access characteristcs are common across various applications [5]. In the baseline memory controller using just static 7-SETs-Writes, there is a high possibility that write requests will be accumulated in the write queue in the *Busy Phase* due to the long latency of 7-SETs-Writes. This situation triggers unwanted write-draining operations (i.e., giving higher priority to write requests over read requests in order to reduce the blockage of the write queue), which in turn degrade the performance of the memory controller. On the other hand, the memory bandwidth is simply wasted in the *Idle Phases* and *Relatively Idle Phases*.

Figure 3 presents an example of our proposed *QnD* scheme. Figure 3(a) shows the baseline system, in which only long 7-SETs-Writes are performed. Figure 3(b) shows the case of our proposed *QnD* scheme, in which the memory controller performs 3-SETs-Writes (i.e., *QnD Writes*) when the memory is busy (e.g., in *Busy Phases*) and then issues extra 7-SETs-Writes refreshes (i.e., *QnD Refreshes*) in *Idle Phases* to mitigate the short retention caused by previously issued *QnD Writes*. By doing so, the write requests have a higher chance of being processed when the memory is busy, and thus the possibility of write-draining operations is reduced. Finally, the overall system performance improves as a result of fewer write-draining operations. In next section, we will discuss in detail of our *QnD* technique.

III. QND SCHEME

A. Hardware Structure

1) QnD Recorder*: QnD Recorder* decides whether a to-be-issued write request needs to be transferred to a *QnD Write* and records address information for the issued *QnD Write*. Considering the hardware overhead, *QnD Recorder* records *QnD* information in *QnD Regions* of 8KB granularity, each of which contains 128 memory blocks. As shown in Figure 4, *QnD Recorder* is organized in a cache-like set-associated manner with a separate tag array and *QnD Information Array*. The format of *QnD Recorder entry* is as shown in the bottom part of Figure 4.

2) QnD Refresh Queue*: QnD Refresh Queue* issues additional refreshes to the MLC PCM array with long latency writes (7-SETs-Writes) to fix the short retention problem caused by *QnD Writes*. The architecture of *QnD Refresh Queue* is similar to the *Write Queue* with two differences: first, the entries of *QnD Refresh Queue* do not contain data fields since the to-be-refreshed memory blocks already contain correct data (but with a short retention time); second, each entry of *QnD Refresh Queue* also has a 4-bit *Decay Counter*, the same as the entries in *QnD Recorder*.

B. Operations of QnD Recorder

The *QnD* scheme requires three operations: *Write Request Issuing*, *QnD Decay*, and *QnD Refresh Generation*.

1) Write Request Issuing: A *QnD* memory controller can issue two write modes: *QnD Writes* and *Normal Writes*. A *QnD* memory controller issues *QnD Writes* whenever it is *necessary* and *possible*:

- *"Necessary"* means that currently the *Write Queue* is under high pressure (the number of queued write requests in *Write Queue* surpasses *QnD_Threshold*) and may become a bottleneck for memory controller performance.
- *"Possible"* means that *QnD Recorder* is able to record the newly generated *QnD Write*. If the write request is hit in a *QnD Entry*, the corresponding *blk_QnD_vector* bit is set as "1." If there is no hit in a *QnD Entry*, the *QnD Recorder* has to initialize an unused *QnD Entry* in the corresponding set. When there are no unused *QnD Entry*, no *QnD Write* can be issued. .

If a memory block previously written with a *QnD Write* is written again with a *Normal Write*, the corresponding *blk_QnD_vector* bit of the *QnD Entry*(if there is a hit) to "0." If the *blk_QnD_vector* become all "0s," the *QnD Entry* will be also invalidated.

2) QnD Decay: To guarantee that data written by *QnD Writes* can be refreshed with *QnD Refreshes* before its retention limit expires, every entry in *QnD Recorder* and *QnD Refresh Queue* contains one extra 4-bit *Decay Counter* field to track the retained time. In every newly initialized *QnD Recorder Entry*, the *Decay Counter* is set to "0" and then increases by 1 every 0.125 seconds. When a *QnD Refresh Request* is sent from *QnD Recorder* to *QnD Refresh Queue*, the value of *Decay Counter* for its corresponding *QnD Recorder Entry* is also copied. Once the *Decay Counter* value reaches 4'b1111, the corresponding entry is identified as *urgent* since some of its data may reach the retention limit.

3) QnD Refresh Request Generation: Before the corresponding short retention data of a *QnD Entry* reach their limit, *QnD Recorder* has to generate *QnD Refresh Requests* to fix the short retention problem, and then sends the request to *QnD Refresh Queue*. When one *QnD Recorder Entry* has a *Decay Counter* value of 4'b1111, it will become *urgent entry* and

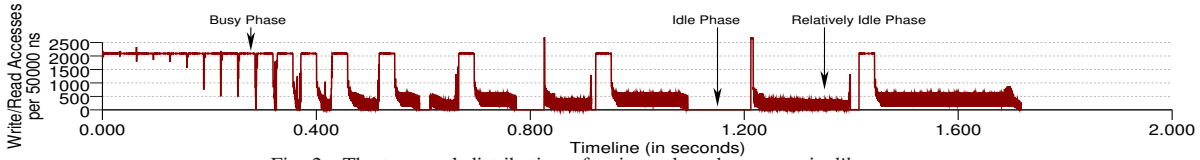

Fig. 2. The temporal distribution of write and read accesses in *libquantum*.

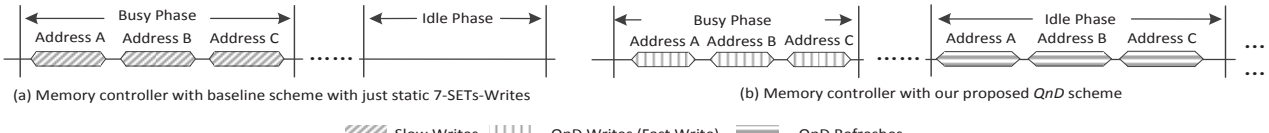

Fig. 3. Motivational comparison between baseline scheme and *"Quick-and-Dirty"* (*QnD*) scheme.

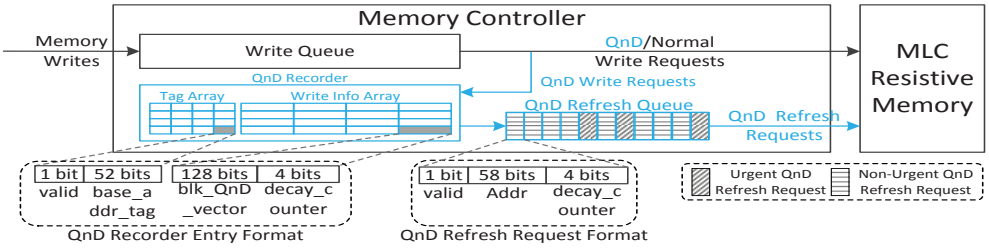

Fig. 4. A high-level view of memory controller with *QnD* implemented.

get the top priority in the *QnD Refresh Request Generation* procedure. The *QnD Refresh Queue* always reserves 4 slots for the *urgent entries*. Besides, to reduce the possibility of hotspot banks, in each cycle, *QnD Recorder* randomly selects the entry to generate *QnD Refresh Request* once *QnD Refresh Queue* has available slots in the left 28 slots. After a *QnD Refresh Request* is generated and passed to *QnD Refresh Queue*, *QnD Recorder* will reset its associated *blk_QnD_vector* bit to "0." If the *blk_QnD_vector* becomes all "0," then the whole *QnD Recorder Entry* gets invalidated.

C. Operations of QnD Refresh Queue

QnD Refresh Queue caches the *QnD Refresh Requests* and then issues them into the memory array at the proper time. Since each *QnD Refresh Request* contains a *Decay Counter* with the value previously copied from its corresponding *QnD Recorder Entry*, in each cycle, *QnD Refresh Queue* increases the value of *Decay Counter* in all *non-urgent* requests by 1. Once the *Decay Counter* value reaches 4b'1111, the request becomes *urgent* one. When issuing the requests, *urgent* requests always have the top priority. Note that, the QnD refresh mechanism provides high reliability even in the worst case. For each bank, at most 80000 QnD memory blocks can be recorded. Sequentially writing these blocks back requires less than 0.1s. Therefore, even if all of them become urgent simultaneously, all blocks can still be written back in time (i.e., 0.25s retention time left).

Note that, the QnD refresh mechanism provides high reliability even in the worst case. For each bank, at most 80000 QnD memory blocks can be recorded. Sequentially writing these blocks back requires less than 0.1s. Therefore, even if all of them become urgent simultaneously, all blocks can still be written back in time.

D. Hardware Overhead

Implementing the *QnD Recorder* and *QnD Refresh Queue* requires little hardware overhead. As shown in Figure 4, each *QnD Recorder* entry requires 185 bits. In our configuration, *QnD Recorder* has 32 sets and 16 ways, which needs 11.56KB of extra storage. *QnD Refresh Queue* contains 32 entries, since each entry requires 63 bits, it totally requires 0.25KB of extra storage. In summary, QnD technique requires less than 12KB of extra storage. This is indeed a trivial overhead compared to the performance/lifetime benefit of *QnD*.

IV. Evaluation

A. Methodology

Our simulation platform is based on GEM5 [6] integrated with NVmain [7], which supports time-accuracy simulation for nonvolatile memory. We simulated an aggressive 8-issue out-of-order Alpha-ISA uniprocessor running a single memory-intensive workload. To evaluate the effectiveness of our scheme on a multi-core system, we also simulate a 4-core CMP running multiple workloads. In our simulation, each core in the simulation contains split 32KB 4-way I/D-cache with 8-MSHR and 256KB 8-way L2 cache with 12-MSHR. Besides, the uniprocessor has 2MB 16-way LLC with 32-MSHR, while the 4-core CMP has a shared 6MB 24-way LLC with 32-MSHR. The hit latency of L1, L2 and Last Level cache are 2-cycle, 12-cycle and 35-cycle respectively.

We simulated the MLC PCM main-memory system running at 400 MHz with 64-bit width bus to evaluate our proposed *QnD* scheme. The memory size for uniprocessor and 4-core CMP are 4GB and 8GB respectively. The write endurance of a single MLC PCM cell is 5×10^6. We assume the memory system uses an effective wear-leveling technique that makes the whole MLC PCM system achieve 95% of the average

TABLE II
WORKLOADS AND THEIR MPKI (MISS PER 1000 INSTRUCTIONS).

For Single-Core			
Workload	MPKI	Workload	MPKI
bwave	8.15	milc	9.83
hmmer	1.58	lbm	22.33
leslie3d	5.47	libquantum	18.04
zeusmp	5.98		

For CMP	
MIX_1	lbm+bwave+zeusmp+milc
MIX_2	milc+libquantum+lbm+leslie3d

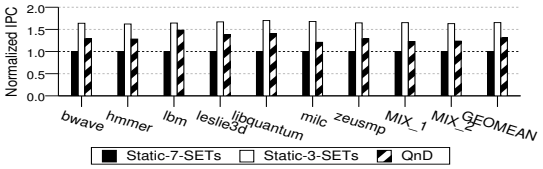

Fig. 5. Normalized performance.

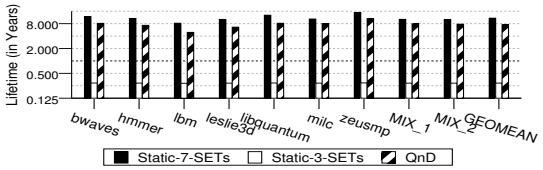

Fig. 6. Memory lifetime.

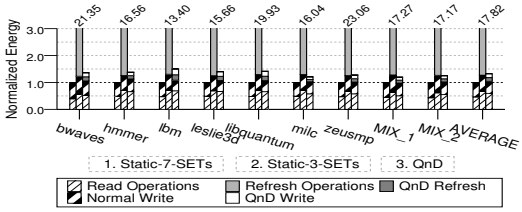

Fig. 7. Normalized energy.

cell lifetime. Each processor core has one memory channel, and each memory channel has 16 banks. The read queue and write queue contain 32entries/channel and 64 entries/channel respectively. We simulate two types of write operation: 3-SETs-Write with 550ns (220-cycle) write latency and 7-SETs-Write with 1150ns (460-cycle) write latency. *QnD* uses 7-SETs-Write as *Normal Write* and *Refresh*, while uses 3-SET-Write for *QnD write*.

We use several memory-intensive applications from the SPEC2006 benchmark suite [8]. For the single-core system, we ran a single application in each experiment, while the CMP system is evaluated with a mixed workload of four different applications. We ran each workload for two seconds. The detail of the workloads are as shown in Table II.

B. Results

1) Performance and Lifetime: We use IPC (Instructions Per Cycle) to evaluate the performance of the evaluated systems. Figure 5 shows the normalized IPC of *Static-3-SETs*, *Static-7-SETs* and *QnD* schemes. We can see that *QnD* achieves on geometric mean 30.9% higher performance than *Static-7-SETs*. Though it seems that the performance of *QnD* is still 20.8% inferior to *Static-3-SETs*, the short lifetime of *Static-3-SETs* makes it impractical for a real system implementation.

Figure 6 presents our memory lifetime comparison. As shown in the figure, *Static-3-SETs* has an unacceptable memory lifetime of 0.29 years. Meanwhile, while *QnD* has a shorter memory lifetime than *Static-7-SETs* (7.58 years vs. 10.87 years), its memory lifetime is within the acceptable range of consumer electronics (the worst being 4.92 years in `lbm`).

2) Energy Consumption: We calculate the energy consumption of different schemes using the parameters given in Table I. Figure 7 shows the results. *Static-3-SETs* requires more than one magnitude (on geometric mean $17.23\times$) higher energy than *Static-7-SETs*, largely due to the fact that it needs to frequently refresh the whole memory. Meanwhile, *QnD* also needs extra energy to issue *QnD Refresh Requests* to fix the retention issues brought on by *QnD Writes*. However, since *QnD Refresh Requests* are relatively limited, only a small amount of extra energy is needed. As a result, *QnD* only requires 33% more memory energy than *Static-7-SETs*. Given with 30.9% performance benefit, this is indeed an affordable expense.

V. CONCLUSION

In this paper, we analyze the write access pattern of common applications and show that there is an imbalanced temporal distribution. Based on this observation, we utilize the tradeoff between write latency and retention in MLC PCM and propose a novel lightweight technique, called *QnD*, to improve MLC PCM write performance. A memory controller with *QnD* dynamically issues write requests in *short-latency, short-retention* mode (i.e., *QnD Writes*) when the write operations become a system bottleneck, and then issues extra *long-latency, long-retention* refreshes to fix the retention problem . Our experiment shows that, compared to a baseline scheme that only uses *long-latency, long-retention* writes, *QnD* improves system performance 30.9% on geometric mean , while the default *QnD* achieves a memory lifetime of 7.58 years on geometric mean.

REFERENCES

[1] L. Zhang, B. Neely, D. Franklin, D. Strukov, Y. Xie, and F. T. Chong,"Mellow writes: Extending lifetime in resistive memories through selective slow write backs, in ISCA, 2016.

[2] M. Zhang, L. Zhang, L. Jiang, Z. Liu, and F. T. Chong, "Balancing Performance and Lifetime of MLC PCM by Using a Region Retention Monitor," in HPCA, 2017.

[3] Y. Choi, I. Song, and M.-H. Park, "A 20nm 1.8v 8gb pram with 40mb/s program bandwidth, in ISSCC, 2012.

[4] K. Kim and S. J. Ahn, Reliability investigations for manufacturable high density pram, in IRPS, 2005.

[5] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, Discovering and exploiting program phases, in MICRO, 2003.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, The gem5 simulator, in SIGARCH Comput. Archit. News, 2011..

[7] M. Poremba and Y. Xie, Nvmain: An architectural-level main memory simulator for emerging non-volatile memories, in VLSI, 2012.

[8] Spec2006 benchmarks, http://www.spec.org/cpu2006/.

[9] Lunkai Zhang, Dmitri B. Strukov, Hebatallah Saadeldeen, Dongrui Fan, Mingzhe Zhang, and Diana Franklin, "SpongeDirectory: flexible sparse directories utilizing multi-level memristors", in PACT, 2014.

[10] Zhaoxia Deng, Lunkai Zhang, Diana Franklin, and Frederic T. Chong, "Herniated Hash Tables: Exploiting Multi-Level Phase Change Memory for In-Place Data Expansion", in MEMSYS, 2015

[11] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T. Chong, "Memory Cocktail Therapy: A General Learning-Based Framework to Optimize Dynamic Tradeoffs in NVMs", in MICRO, 2017.