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

ABSTRACT

MLC PCM provides high-density data storage and extended data retention; therefore it is a promising alternative for DRAM main memory. However, its low write performance is a major obstacle to commercialization. One opportunity for improving the latency of MLC PCM writes is to use fewer SET iterations in a single write. Unfortunately, this comes with a cost: the data written by these short writes have remarkably shorter retentions 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 show that the write operations are bursty in nature, that is, during some time intervals the memory is subject to a large number of writes, while during other time intervals there hardly any memory operations take place. Based on this observation, we propose *Quick-and-Dirty (QnD)*, a lightweight scheme to improve the performance of MLC PCM. When the write performance becomes the system bottleneck, *QnD* performs some write operations using the short-latency, short-retention write mode. Then, when the memory system is relatively quiet, *QnD* uses idle-memory intervals to refresh the data written by short-latency, short-retention writes in order to mitigate the short retention problem.

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.

KEYWORDS

MLC PCM; Dynamic tradeoff; performance

ACM Reference format:

. 2017. Quick-and-Dirty: Improving Performance of MLC PCM by Using Temporary Short Writes. In *Proceedings of ACM International Conference of Supercomputing, Chicago, Illinois USA, June 2017 (ICS'17),* 10 pages.

DOI: 10.475/123_4

1 INTRODUCTION

Phase Change Memory (PCM) belongs to the resistive-memory family [14, 26, 29, 30] which represents digital values with different resistances. Since it provides high integration density and low power consumption and has no volatility, PCM is considered to be a potential alternative technology for DRAM [15, 27, 33]. Additionally, when its resistance is carefully controlled , PCM allows the storage of multiple bits in a single cell. PCM that incorporates enhancement technique is called Multi-Level Cell (MLC) PCM. 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. However, as will be discussed in Section 2, this comes at the cost of shorter data retention. For example, according to Table 1, 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.

The rest of this paper is organized as follows. Section 2 introduces the *write latency vs. retention* trade-off in detail. In Section 3, we provide the motivation behind our research. Then we introduce the *QnD* technique in detail in Section 4. The experimental methodology and results analysis of our experiment are presented respectively in Sections 5 and 6. Section 7 presents related work, and Section 8 concludes the paper.

2 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 between retention time and write latency, we modelled a MLC PCM cell in detail using the parameters

ICS'17, Chicago, Illinois USA

DOI: 10.475/123_4

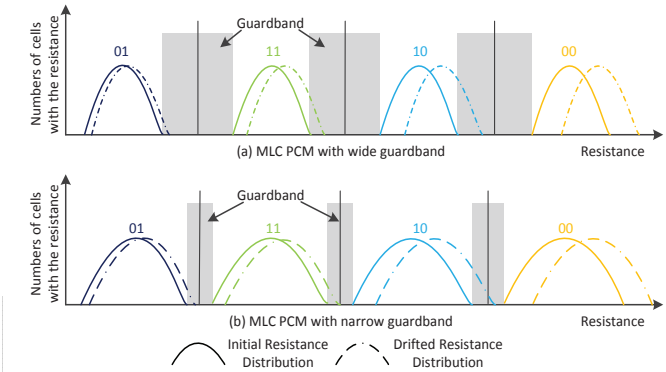

Figure 1: Resistance drift problem in MLC PCM with guardband technique.

Table 1: The trade-off between latency and retention of MLC PCM write operations.

Write Type	Latency (in ns.)	Retention (in seconds)	Current (in μA)	Normalized 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

from the latest demonstrated 20nm PCM chip [6]. The model includes the write-latency/write-retention model [11, 16] , the resistance-drifting model [2], the process-variation model [32] and the write-energy/cell-endurance model [13]. In our model, a MLC PCM write has a RESET operation followed by several SET operations. A RESET operation needs 100ns and 50μA, while a SET operation needs 150ns. SETs are done by multiple iterations. Varying numbers of iterations (from 3 to 7) require different current amplitudes (from 42uA to 30uA). The model results are presented in Table 1. We can see that the retention time decreases superlinearly with the reduction of SET iterations: a 7-SETs-write achieves a retention time of 3054.9 seconds, while a 3-SETs-write only 2.01 seconds retention. On the other hand, the longer retention time does come with a cost: a 7-SETs-write costs more than double the latency than a 3-SETs-write (1150ns vs. 550ns).

Since PCM endurance is dominated by RESET operations and SET operations have little endurance impact[13], in our model we assume all write operations achieve the same endurance irrespective of the number of their SET iterations.

3 MOTIVATION

In this section, we first analyze the temporal behavior of MLC PCM write operations and then introduce the notion of our proposed scheme. Figure 2 shows the time pattern of write operations in our baseline system (described in Section 5) running benchmark `libquantum`. We can see that, according to the behavior of `libquantum`, the temporal distribution of its write operations are highly uneven. For example, during the *Busy Phase* from 0s to 0.37s there are a large number of read/write requests. However, during the *Idle Phase* from 1.1s to 1.2s, there are no memory operations at all. Also, there are many *Relatively Idle Phases* (e.g., 1.23s to 1.35s) which have relatively few memory operations. 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.

4 SCHEME

In this section, we introduce the detailed *QnD* technique. Figure 4 presents the high-level view of a memory controller that employs the *QnD* technique. The additional structures and data-paths are highlighted in blue. There are two additional structures needed in our *QnD* scheme: *QnD Recorder* and *QnD Refresh Queue*.

- *QnD Recorder* decides whether a to-be-issued write request needs to be transferred to a *QnD Write*. If so, it records address information for the issued *QnD Write*. *QnD Recorder* also checks its recorded *QnD* information and fixes the the short retention problem of previous *QnD Writes* by sending *QnD Refresh Requests* to *QnD Refresh Queue*.
- *QnD Refresh Queue* issues the *QnD Refresh Requests* generated by *QnD Recorder*.

In the rest of this section, we describe the hardware structure and operations of *QnD* in detail.

4.1 Hardware Structure

4.1.1 QnD Recorder. Ideally, *QnD Recorder* should record the *QnD* information in the granularity of memory blocks (i.e., 64 bytes). However, this requires nontrivial storage. To reduce the hardware overhead, we record *QnD* information in *QnD Regions* of 8KB granularity, each of which contains 128 memory blocks. Section 6 shows that such a design choice achieves a nice balance between effectiveness and hardware storage. As shown in Figure 4, *QnD Recorder* is organized in a set-associated manner with a separate tag array and *QnD Information Array*, just like a cache. A *QnD Recorder entry* includes following fields:

- *Valid Bit (1 bit)*, which indicates whether the entry is in use or not.
- *Address Tag (52 bits)*, which records the base address information of its associated *QnD Region*.

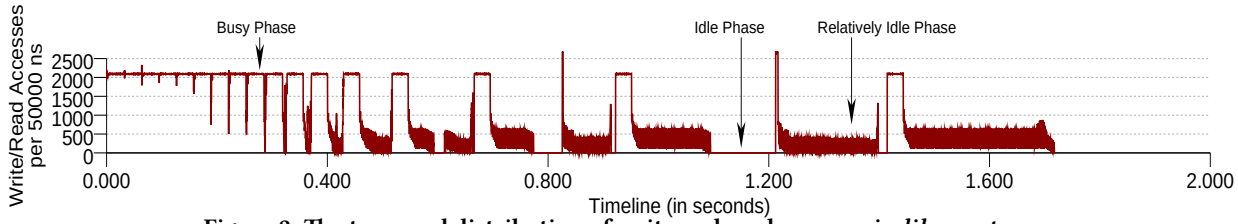

Figure 2: The temporal distribution of write and read accesses in *libquantum*.

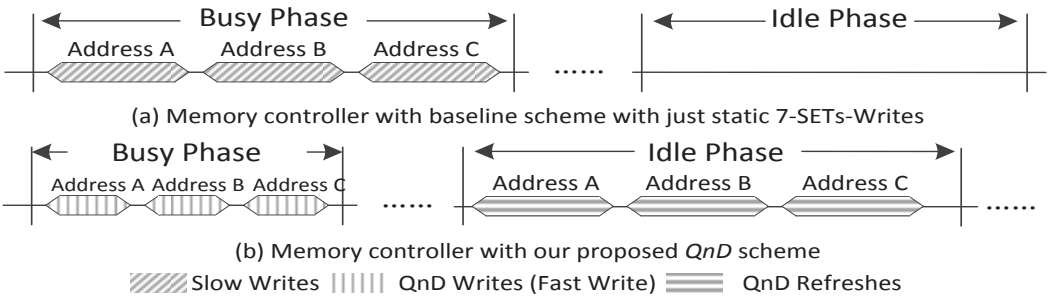

Figure 3: Motivational comparison between baseline scheme and *"Quick-and-Dirty"* (*QnD*) scheme.

- *QnD Vector (128 bits)*, each bit of which indicates whether its associated memory block within the *QnD Region* was written with *QnD Writes*. If a memory block is written with *QnD Writes*, the corresponding bit is set to 1; otherwise, the bit is set to 0.
- *Decay Counter (4 bits)*, which indicates how much time has passed since the first write operation to the corresponding *QnD Region*. A *Decay Counter* of 4'b1111 means this is an urgent *QnD Recorder entry* that might reach its retention limit soon; therefore it is a must to handle the retention issue of this *QnD Region* by issuing *QnD Refresh Requests* as soon as possible.

4.1.2 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*.

4.2 Operations of QnD Recorder

The *QnD* scheme requires a modified *Write Request Issuing* operation. It also needs three additional operations: *QnD Decay*, and *QnD Refresh Generation* and *Issuing*.

4.2.1 Write Request Issuing. Recall that, unlike conventional memory controllers that just issue one write mode (i.e., *Normal Writes*), a *QnD* memory controller can issue two write modes: *QnD Writes* and *Normal Writes*. As shown in the top portion of Figure 5, a *QnD* memory controller issues *QnD Writes* whenever it is *necessary* and *possible*:

- *"Necessary"* means that currently the *Write Queue* is under high pressure and may become a bottleneck for memory controller performance. A new *QnD_Threshold* metric is used to measure the pressure of *Write Queue*. If the number of queued write requests in *Write Queue* surpasses *QnD_Threshold*, it is necessary to use *QnD Writes* to relieve the pressure of *Write Queue*.
- *"Possible"* means that *QnD Recorder* is able to record the newly generated *QnD Write*. If the write request is hit in a *QnD Entry*, *QnD Recorder* has no problem recording the incoming *QnD Write*—all it needs to do is set the corresponding *blk_QnD_vector* bit to "1." If there is no hit in a *QnD Entry*, the *QnD Recorder* has to initialize an unused *QnD Entry* in the corresponding set. In cases that there are no unused *QnD Entry*, the write request cannot be transferred to *QnD Write*.

If a memory block previously written with a *QnD Write* is written again with a *Normal Write*, its short retention problem gets fixed and it does not need an extra *QnD Refresh Request*. Therefore, upon the issuance of each *Normal Write* request, the *QnD Recorder* also needs to update itself accordingly to remove the corresponding *QnD* information. This is done by simply resetting the corresponding *blk_QnD_vector* bit of the corresponding (if there is a hit) *QnD Entry* to "0." If the *blk_QnD_vector* become all "0s," the *QnD Entry* will be also invalidated.

4.2.2 QnD Decay. In order to guarantee that data written by *QnD Writes* can be refreshed with *QnD Refreshes* before its retention limit expires, *QnD* tracks the retained time of the data written by *QnD Writes*. To do this, we add one extra 4-bit *Decay Counter* field in every entry in *QnD Recorder* and *QnD Refresh Queue*.

When a *QnD Recorder Entry* gets initialized, its associated *QnD Recorder* value is set to "0." When a *QnD Refresh Request* is sent

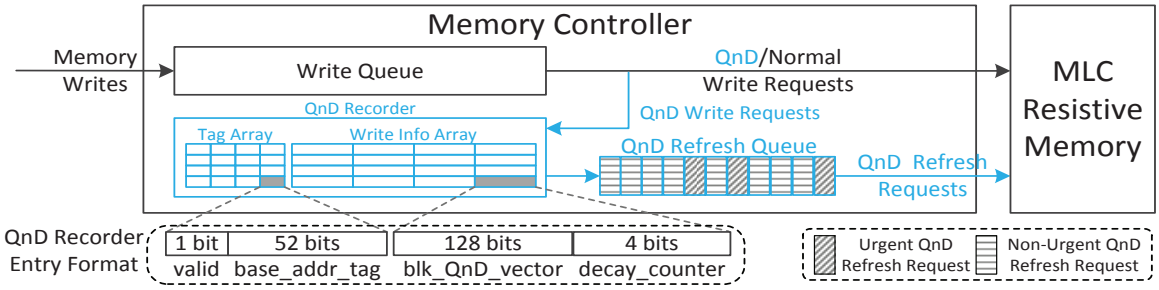

Figure 4: A high-level view of memory controller with *QnD* implemented.

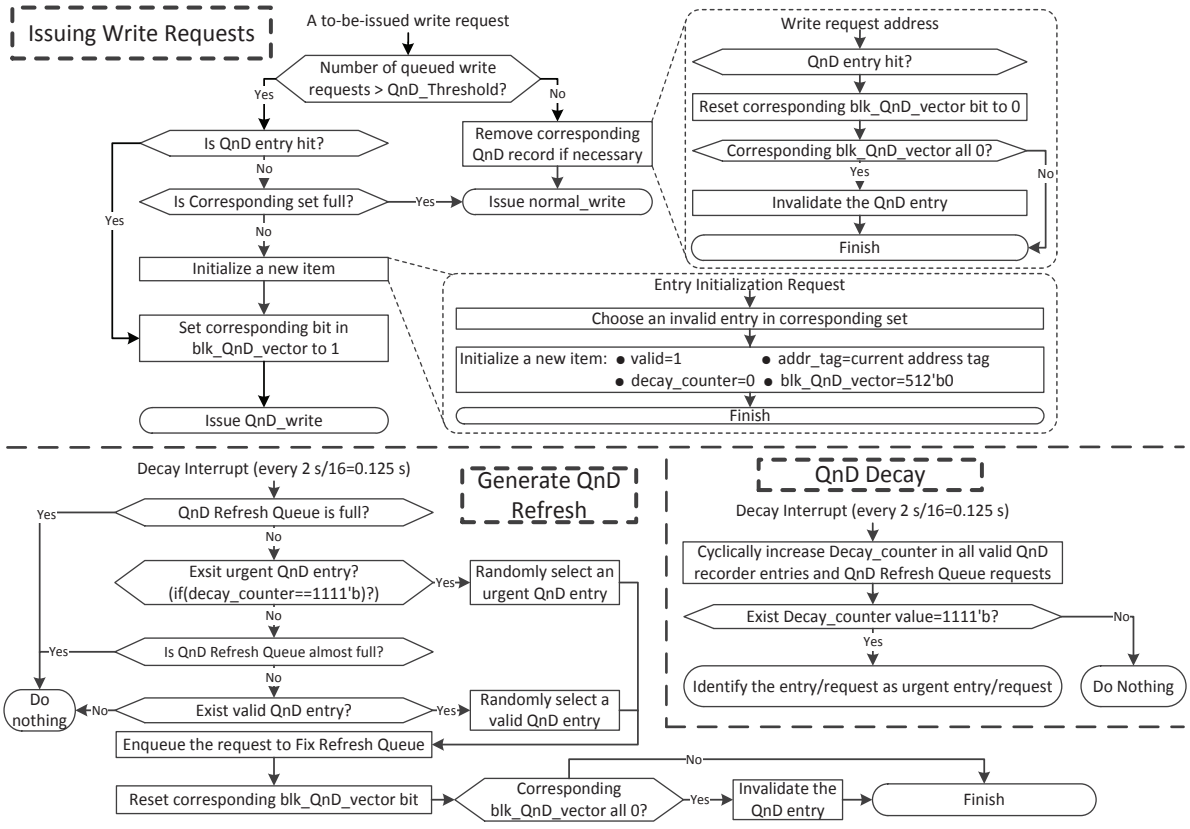

Figure 5: *QnD Recorder* operations: *Issuing Write Request, Generating QnD Refresh* and *QnD Decay*.

from *QnD Recorder* to *QnD Refresh Queue*, the value of *Decay Counter* for its corresponding *QnD Recorder Entry* is also copied to the new *QnD Refresh Queue* entry. A *QnD_Decay_Interrupt* is generated every 0.125 seconds (1/16 of the two-second retention time of *QnD Writes*). Upon each *QnD_Decay_Interrupt*, *Decay Counter* for all the entries is increased by 1. Once the *Decay Counter* value reaches 4'b1111, the corresponding entry (whether in *QnD Recorder* or *QnD Refresh Queue*) is identified as *urgent* since some of its data may reach the retention limit.

4.2.3 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. The generated *QnD Refresh Requests* will be then issued by *QnD Refresh Queue*.

Recall that in Section 4.2.2, we classify the *QnD Recorder Entries* into two categories: *urgent* ones with a *Decay Counter* value of 4'b1111, and *non-urgent* ones with a *Decay Counter* value less than 4'b1111. The *QnD Entries* will expire soon, so they are given top priority in the *QnD Refresh Request Generation* procedure. Such top priority involves two privileges:

- During each refresh generation procedure, *QnD Recorder* always considers *urgent* entries before *non-urgent* ones.

- *QnD Recorder* can generate a *QnD Refresh Request* for an *urgent* entry as long as the *QnD Refresh Queue* is not *full*; however, it is able to generate a request for a *non-urgent* entry only when the *QnD Refresh Queue* is not close to be full (i.e., less than 87.5% percent of the queue entries are occupied). That is, *QnD Refresh Queue* always reserves slots for the requests of *urgent* entries.

Note that, besides prioritizing *urgent* entries over *non-urgent* ones, the selection of *QnD Entries* during *QnD Refresh Request Generation* operations is completely random. For example, an entry with a *Decay Counter* of 4'b1110 does not have higher priority than an entry with a *Decay Counter* of 4'b0000. We randomize the entry selection to more evenly distribute the *QnD Refresh Requests* into different memory banks, thus reducing the possibility of hotspot banks.

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.

4.3 Operations of QnD Refresh Queue

Compared to *QnD Recorder*, *QnD Refresh Queue* has a relatively simpler function: it just caches the *QnD Refresh Requests* from *QnD Recorder* and then issues them into the memory array at the proper time. Only two details need to be managed.

- *Decay Counter.* In order to differentiate *urgent* and *non-urgent* requests, we also include a *Decay Counter* field in every *QnD Refresh Request* entry of *QnD Refresh Queue*. When a *QnD Refresh Request* is generated from a *QnD Recorder Entry*, it copies the value of the *Decay Counter* field of *QnD Recorder Entry* to its own *Decay Counter* field. Since a *QnD Refresh Request* may transfer from *non-urgent* to *urgent* status during its queued time, the *Decay Counters* of all the valid requests in *QnD Refresh Queue* also get increased by 1 upon every *QnD_Decay_Interrupt*.
- *Request Priority.* To avoid performance penalties, *non-urgent QnD Refresh Requests* are handled with the lowest priority (i.e., lower than read and write requests). On the other handle, *urgent QnD Refresh Requests* are handled with the highest priority to prevent the data errors caused by retention violations. However, in our experiment, most of the *QnD Refresh Requests* issued are *non-urgent*; therefore there is a trivial number of performance penalties caused by *QnD Refresh Requests*.

4.4 Hardware Overhead

Implementing the *QnD Recorder* and *QnD Refresh Queue* requires little hardware overhead. Table 2 shows the implementation details of both structures:

- *QnD Recorder* is a set-associative structure. Each *QnD Recorder* entry has a 1-bit *valid* field, a 52-bit *base_addr_tag* field, a 128-bit *blk_QnD_vector* field and a 4-bit *decay_counter* field. In total, a *QnD Recorder* entry requires 185 bits. In our default configuration, *QnD Recorder* has 32 sets and 16 ways. As a result, it needs 11.56KB of extra storage.
- *QnD Refresh Queue* contains 32 entries, including four reserved entries for the *urgent QnD Refresh Requests*. Each *QnD Refresh Queue* entry contains three fields: a 1-bit *valid*, a 58-bit *addr* and a 4-bit *decay_counter*. As a result, *QnD Refresh Queue* requires 0.25KB of extra storage.

In summary, by default, our QnD technique requires less than 12KB of extra storage. This is indeed a trivial overhead compared to the performance/lifetime benefit of *QnD*.

5 METHODOLOGY

As is shown in Table 3, our simulation platform is based on GEM5 [3] integrated with NVmain [21], which supports time-accuracy simulation for nonvolatile memory. We simulated an aggressive 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.

We simulated the MLC PCM main-memory system to evaluate our proposed *QnD* scheme. The detailed memory system parameters are shown in Table 2. 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 cell lifetime. Each processor core has one memory channel, and each memory channel has 16 banks.

By default, *QnD Recorder* has 16 ways and 32 sets. And each *QnD Recorder* entry has a *blk_QnD_vector* of 128 bits: that is, a *QnD Recorder* entry covers a memory region of 128 cache lines (i.e., 8KB). As a result, *QnD Recorder* has a memory coverage of 4MB. And *QnD Refresh Queue* has 32 entries. As is discussed in Section 4.4, such a default configuration has a trivial hardware overhead which is less than 12KB.

Table 4 presents the write scheme used in our experiments. We simulated five static write schemes and compare the results against our proposed *QnD* scheme. The *N-SETs-Static-Write* indicates the scheme that globally uses *N-SETs-Write*. For *QnD*, some write requests are adaptively fulfilled with *3-SETs-Write* and then fixed with *7-SETs-Write*. Note that, since all regions written with *fast write* are refreshed in two seconds, we did not simulate in detail the performance overhead of global refresh and just consider it statically in a lifetime estimation.

We use several memory-intensive applications from the SPEC2006 benchmark suite [1]. 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 5.

6 RESULTS

In this section, we first show the effectiveness of our proposed *QnD* scheme by comparing *QnD* with *Static-3-SETs* and *Static-7-SETs* regarding various aspects, including *performance*, *memory lifetime*, and *energy consumption*. To show why *QnD* works, we also present the details of *accumulated wear* and *write drain time*. We also compare *QnD* with other static schemes (i.e., *Static-4-SETs*, *Static-5-SETs* and *Static-6-SETs*). At the end, we will conduct sensitivity

Table 2: MLC PCM memory subsystem parameters.

	Single-Core	CMP
Mem. Size	4GB	8GB
Freq.	400 MHz	
Bus Width	64-bit	
# of Channels	1	4
# of Banks	16 per channel	
Read Queue	32 entries per channel, medium-high priority.	
Write Queue	64 entries per channel, medium-low priority.	
QnD Recorder	32-set, 16-way Each entry requires 185-bit storage: – 1-bit *valid* – 52-bit *base_addr_tag* – 128-bit *blk_QnD_vector* – 4-bit *decay_counter* Total memory coverage is 4MB. Total storage requirement is 11.56KB.	
QnD Refresh Queue	32 entries per channel Each entry requires 63-bit storage: – 1-bit *valid* – 58-bit *addr* – 4-bit *decay_counter* Total storage requirement is 0.25KB. Two request priorities: –*Urgent QnD Refresh Request* with high priority –*Non-Urgent QnD Refresh Request* with low priority	
Page Policy	open page	
Row Size	16KB	
Row Buf Size	1KB	
tRCD	48 cycles (120ns)	
tCAS	1 cycle (2.5ns)	
tFAW	50ns	
tWP (write pulse time)	3-SETs-Writes: 220 cycles (550 ns); 4-SETs-Writes: 280 cycles (700 ns); 5-SETs-Writes: 340 cycles (850 ns); 6-SETs-Writes: 400 cycles (1000 ns); 7-SETs-Writes: 460 cycles (1150 ns).	
Endurance	5×10^6 writes	
Misc	Assuming an effective wear leveling scheme (e.g., [25]), that makes whole memory achieve an average of 95% of cell lifetime. Write Pausing [24] supported. Using Write-through Policy (bypassing row buffer).	

studies of the agressiveness, memory coverage and granularity of *QnD* technique.

6.1 Performance and Lifetime

We use IPC (Instructions Per Cycle) to evaluate the performance of the evaluated systems. Figure 6 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. Also, since we do not simulate the global refreshes in detail (as discussed in Section 5), the performance of *Static-3-SETs* will actually be worse than what we report here because of the penalty of global refresh operations.

Figure 7 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

Table 3: Processor parameters.

	Single-Core	CMP
Freq.	2GHz	
# Cores	1	4
Core	Alpha ISA, OoO, 8 issue	
L1 Caches	split 32KB I/D-cache/core, 4-way, 2-cycle hit latency, 8-MSHR	
L2 Caches	256KB/core, 8-way, 12-cycle hit latency, 12-MSHR	
L3 Cache (LLC)	2MB, 16-way, 35-cycle hit latency, 32-MSHR	shared 6MB, 24-way, 35-cycle hit latency, 32-MSHR

Table 4: Write schemes.

Static-7-SETs	globally using 7-SETs-Writes, global refresh interval: 3054 seconds.
Static-6-SETs	globally using 6-SETs-Writes, global refresh interval: 991 seconds.
Static-5-SETs	globally using 5-SETs-Writes, global refresh interval: 104 seconds.
Static-4-SETs	globally using 4-SETs-Writes, global refresh interval: 24 seconds.
Static-3-SETs	globally using 3-SETs-Writes, global refresh interval: 2 seconds.
QnD	our proposed scheme, selectively using 3-SETs-Writes and 7-SETs-Writes, using 7-SETs-write to refresh the regions previously written with 3-SETs-Write, global refresh interval: 3054 seconds.

Table 5: Workloads and their MPKI (Miss per 1000 Instructions).

Workload	MPKI	Workload	MPKI
For Single-Core			
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		

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 application `lbm`).

In Sections 6.4 and 6.3, we will discuss in detail the reasons for the performance and lifetime benefits of *QnD*.

6.2 Energy Consumption

We calculate the energy consumption of different schemes using the parameters given in Table 1. Figure 8 shows the results. *Static-3-SETs* requires more than one magnitude (on geometric mean 17.23×) 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.

6.3 Accumulated Wear

The reason why *QnD* largely improves the memory lifetime (compared to *Static-3-SETs*) lies in the fact that this scheme introduces

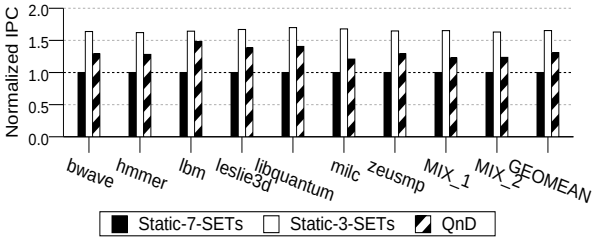

Figure 6: Normalized performance.

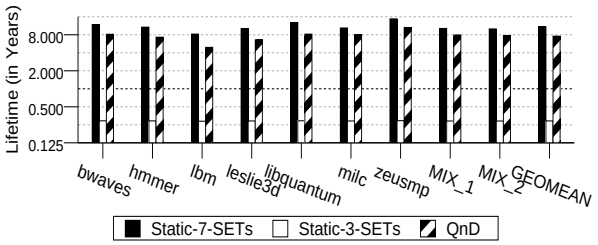

Figure 7: Memory lifetime.

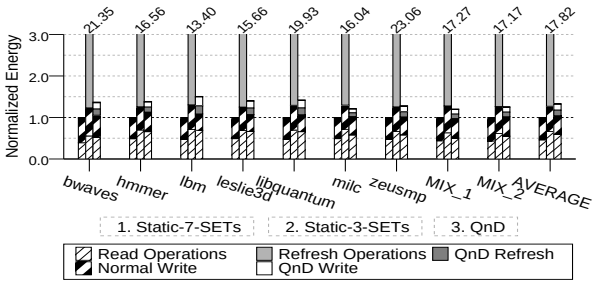

Figure 8: Normalized energy.

much less wear. Figure 9 shows the accumulated wear of three evaluated schemes. Unlike *Static-3-SETs*, which blindly refreshes the whole memory every two seconds, *QnD* issues a relatively small number of *3-SETs-Writes* (i.e., *QnD Writes*) and only issues additional refresh requests (i.e., *QnD Refresh Requests*) to fix the retention of the memory blocks written by *QnD Writes*. As a result, compared to the wear of *3-SETs-Writes*, the wear of *QnD* is remarkably reduced. As a result, *QnD* introduces just 16.1% more wear than *Static-7-SETs*. In comparison, *Static-3-SETs* produces an unacceptable amount of wear 37.87× more than *Static-7-SETs*.

6.4 Write-Drain Time

Write drain happens when the occupancy rate of the *Write Queue* is higher than a threshold (usually full). During a write-drain period, the write requests have higher priority than read requests. As discussed in Section 3, write drain is the reason MLC PCM writes can degrade system performance. Therefore, the amount of time used in write drain becomes a direct metric to evaluate the effectiveness of write schemes in enhancing performance. Figure 10 shows the write-drain time of *Static-3-SETs*, *Static-7-SETs*

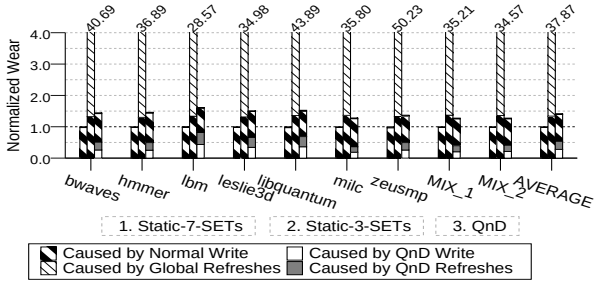

Figure 9: Accumulated wear.

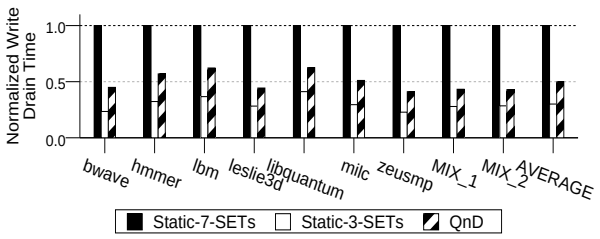

Figure 10: Normalized write-drain time comparison

and *QnD*. We can see that *Static-7-SETs* requires 3.45× more write-drain time than *Static-3-SETs*, and this explains why *Static-3-SETs* outperform *Static-7-SETs*. Meanwhile, *QnD* only requires 1.68× more write-drain time than *Static-3-SETs*. As a result, *QnD* has considerably better performance than *Static-7-SETs* but does not outperform *Static-3-SETs*.

6.5 Comparing QnD to Other Static Schemes

In the previous evaluation, we compared *QnD* with just *Static-3-SETs* and *Static-7-SETs* because they share the same write types (*3-SETs-Writes* and *7-SETs-Writes*). In this subsection, we compare *QnD* with various static write schemes. Figures 11(a) and 11(b) show the results. Not surprisingly, for static schemes, the performance degrades and lifetime improves as more SETs are used in the write operation. We can see that, for most benchmarks such as `bwave` and `lbm`, *QnD* achieves considerably better performance than the static schemes that can provide acceptable memory lifetime (i.e., *Static-7-SETs*, *Static-6-SETs* and *Static-5-SETs*). Overall, our *QnD* achieves a nice balance between performance and lifetime.

On the other hand, we acknowledge that the effectiveness of *QnD* is not ideal for some applications. For example, in benchmark `hmmer`, *QnD* achieves a similar performance to but a slightly shorter lifetime than *Static-6-SETs*. However, we argue that *QnD* is inferior in this case not because the notion of *QnD* is ineffective, but simply because the write types chosen for *QnD* are not perfect. We statically choose *3-SETs-Writes* and *7-SETs-Writes*, respectively as *QnD Writes* and *Normal Writes*, and such a choice is not the most effective for benchmark `hmmer`. If we choose *6-SETs-Writes* as *Normal Writes*, the *QnD* will achieve a better performance than the *Static-6-SETs* scheme and still deliver an acceptable memory

lifetime. In our future work, we will conduct research on how to properly choose the write types for the *QnD* scheme.

6.6 Sensitivity to Aggressiveness of QnD

Recall that in Section 4.2.1 we use *QnD_Threshold* to determine whether to use *QnD Writes*. In fact, we can control the aggressiveness of the *QnD* scheme by altering *QnD_Threshold*. A lower *QnD_Threshold* makes the system generates *QnD Writes* more aggressively. As a result, the system performance improves. However, it also means more *QnD Refresh Requests* need to be issued, which degrades the lifetime of MLC PCM. In this subsection, we quantitatively show how the value of *QnD_Threshold* affects the performance and lifetime of MLC PCM.

As shown in Figure 12, as *QnD_Threshold* increases from 16 to 48, the performance degenerates and the memory lifetime improves. Here we choose a *QnD_Threshold* of 32 as the default configuration. If system users care about performance more than memory lifetime, the configuration of *QnD_Threshold* can be set as 16 or 24, and the system performance can be improved on geometric mean by 6.13% and 3.67%, respectively. On the other hand, if system users want longer memory lifetime, they can increase QnD Threshold. For example, memory lifetime can be improved 7.58% on geometric mean by setting *QnD_Threshold* to 48.

6.7 Sensitivity to Memory Coverage of QnD

Recall that *QnD Recorder* will not be able to convert a *Normal Write* to a *QnD Write* if *QnD Recorder* cannot record the new *QnD Write* (See Figure 5). A larger memory coverage of *QnD Recorder* indicates that it is able to record more *QnD Write* addresses, therefore such an unwanted service rejection situation is less likely to happen. Therefore, the memory coverage of *QnD Recorder* entries plays an important role in its effectiveness. As is shown in Figure 13, we simulate the *QnD* scheme with the memory coverage of *QnD Recorder* from 2MB (256 entries) to 16MB (2048 entries). Not surprisingly, the system performance gets improved as the memory coverage of *QnD Recorder* increases. However, since *QnD* sacrifices some lifetime for the performance by issuing additional *QnD Refreshes*, the lifetime also slightly degrades at the same time. For example, a 8MB-coverage *QnD* achieves 7.81% better performance than the default 4MB-coverage *QnD*, but it also results in 7.01% shorter memory lifetime. Also, increasing memory coverage in *QnD Recorder* also incurs higher hardware overhead. Overall, in our system configuration, a default 4MB memory coverage of *QnD Recorder* achieves a nice balance among performance, lifetime and hardware overhead.

6.8 Sensitivity to Granularity of QnD

As is discussed in Section 4, there is a trade-off in choosing the granularity of *QnD Recorder* entries. On one hand, a larger granularity can help reducing the hardware overhead by removing some of the address tag storage; on the other hand, however, it also increases the possibility of entry conflict in *QnD Recorder* and thus increases the possibility of service rejection. Here we fix the memory coverage of *QnD Recorder* as 4MB, and then simulate a wide range of granularities of *QnD Recorder*, from 1 memory block (64B) to 2048 memory blocks (128KB). The performance and lifetime results are shown in Figure 14. We can see that, though *QnD* achieves better effectiveness in the extreme case of 64B granularity, it requires unacceptably large hardware overhead (0.45 MB). In another extreme case of 128KB granularity, *QnD* is much less effective. As is shown in the figure, a granularity of 8KB is a sweet spot of low hardware overhead and high effectiveness.

7 RELATED WORK

Long write latency is a major disadvantage of nonvolatile memories (NVM), especially for MLC NVM. Various schemes have been proposed to overcome this shortcoming.

One effective approach is to cancel (i.e., *Write Cancellation*) and pause (i.e., *Write Pausing*) the ongoing write operation to give the preemption to an incoming read [24, 28]. One shortcoming of such an approaches is that it might increase the possibility of write drain by increasing the queued time of the write requests [30]. Nevertheless, in our work, we adopt *Write Cancellation* in all of our experiment simulations

Several solutions have been proposed to improve the parallelism of NVM write operations so that the long write latency can be hidden by other write operations[5, 9, 10]. In addition, some solutions add a DRAM-based or SRAM-based buffer in front of NVM so that the write operation to NVM can be hidden by the filtering effect of the buffer[7, 14, 26, 31]. When it comes to MLC NVM, some proposals avoid paying the unnecessary long read and write latency penalty of MLC NVM by dynamically switching the memory regions between SLC and MLC modes [7, 8, 23, 31].

Several prior proposals [12, 17, 18, 20, 22] also use the *write latency vs. retention* trade-off in MLC PCM:

- CDDW[17], WMALT[22] and MMAS[20] use a DAG-based compiler-assistant technique to predict the data lifetime and then use proper write modes for different data. However, these techniques have two limitations. First, they predict the data lifetime by using Direct Acyclic Graph (DAG), which works well for static programs but is hardly useful for dynamic programs. Second, these proposals do not work well in the presence of a multi-level cache system because the filtering effect of caches makes the DAG-based data lifetime prediction inaccurate. Our proposed *QnD* technique does not have such limitations.
- NVM Duet [18] can adaptively make a memory region work as normal memory (low retention mode) or persistent storage [4, 19] (long retention mode) by adopting different write modes. However, such a technique only supports coarse granularity in which a whole program can only choose a single write mode. In addition, this scheme requires nontrivial modification of the operating system. On the contrary, our *QnD* technique supports multiple write modes inside a single program and does not require modification of the operating system.
- Amnesic Cache [12] is proposed for file caches. In this technique, data are always written with short-retention, short-latency writes first and then some frequently accessed data will be written by long-retention, long-latency writes

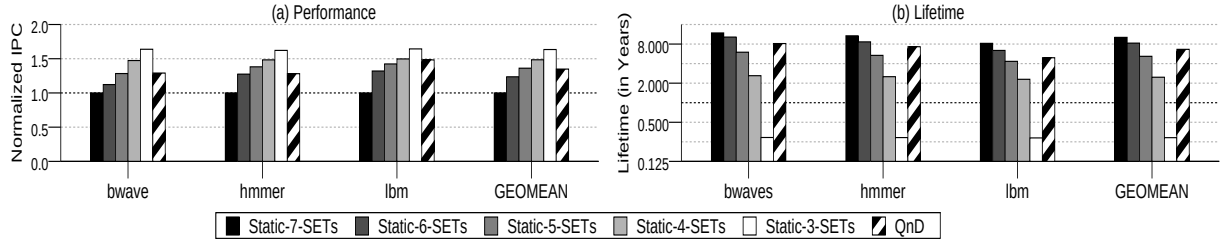

Figure 11: IPC and lifetime comparison with other static schemes.

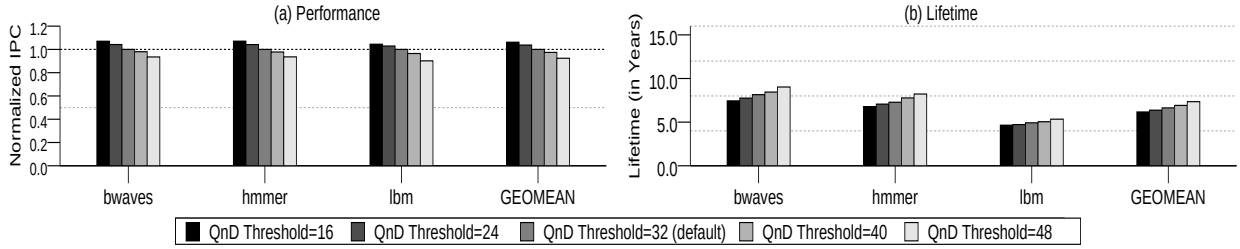

Figure 12: Sensitivity to Aggressiveness of *QnD Recorder*.

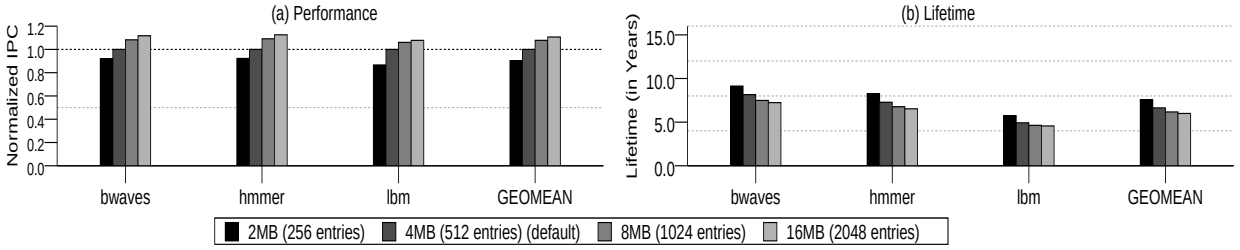

Figure 13: Sensitivity to Memory Coverage of *QnD Recorder*.

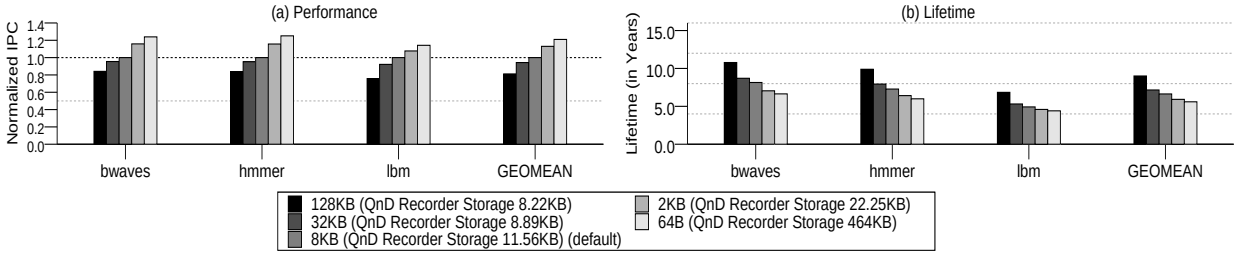

Figure 14: Sensitivity to Granularity of *QnD Recorder*.

again. Meanwhile, the unpromoted data will be evicted when their retention time expires. Such a technique works for cache-like structures in which the data can be evicted instead of refreshed. However, it cannot work for general-purpose main-memory systems in which all the data should be properly maintained.

8 CONCLUSION

In MLC PCM, there exists a trade-off between write latency and retention time. That is, we can reduce the latency of MLC PCM write operations by adopting fewer SET iterations within the write operations. However, this in turn decreases the retention of the written data. Several prior proposals use such a trade-off to improve the write performance of MLC PCM. However, none of these proposals are suitable for the general-purpose computing system, as discussed in Section 7.

In this paper, we analyze the temporal write behaviors of common applications and show that the temporal-access pattern is imbalanced: in some phases there are a large number of write requests, while in other phases, hardly any write requests occur. Based on this observation, we propose a novel lightweight technique, *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. As a result system performance is greatly improved due to the improved write performance during the busy memory phases. Then, the *QnD* memory controller will issue extra *long-latency, long-retention* refreshes to fix the retention problem created by *QnD Writes*.

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. On the other hand, the memory lifetime of *QnD* is still within the acceptable range—the default *QnD* achieves a memory lifetime of 7.58 years on geometric mean. We also show that we can control the aggressiveness of *QnD* by changing its *QnD_Threshold*. For example, if we change *QnD_Threshold* from the default value of 32 to 16, we can improve the system performance by 6.1% at the expense of a shorter memory lifetime (6.90 years vs. 7.58 years).

REFERENCES

[1] SPEC2006 benchmarks. http://www.spec.org/cpu2006/. (????).

[2] Manu Awasthi, Manjunath Shevgoor, Kshitij Sudan, Bipin Rajendran, Rajeev Balasubramonian, and Viji Srinivasan. 2012. Efficient Scrub Mechanisms for Error-Prone Emerging Memories. In *HPCA*.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7.

[4] Adrian M Caulfield, Todor I Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing safe, user space access to fast, solid state disks. *ACM SIGARCH Computer Architecture News* 40, 1 (2012), 387–400.

[5] Sangyeun Cho and Hyunjin Lee. 2009. Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy and endurance. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 347–357.

[6] Youngdon Choi, Ickhyun Song, and Mu-Hui Park. 2012. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *ISSCC*.

[7] Zhaoxia Deng, Lunkai Zhang, Diana Franklin, and Frederic T. Chong. 2015. Herniated Hash Tables: Exploiting Multi-Level Phase Change Memory for In-Place Data Expansion. In *Proceedings of the 2015 International Symposium on Memory Systems*. 247–257.

[8] Xiangyu Dong and Yuan Xie. 2011. AdaMS: Adaptive MLC/SLC phase-change memory design for file storage. In *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*. 31–36.

[9] Andrew Hay, Karin Strauss, Timothy Sherwood, Gabriel H Loh, and Doug Burger. 2011. Preventing PCM banks from seizing too much power. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 186–195.

[10] Lei Jiang, Youtao Zhang, Bruce R Childers, and Jun Yang. 2012. FPB: Fine-grained power budgeting to improve write throughput of multi-level cell phase change memory. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 1–12.

[11] Chul-Moon Jung, Eun-Sub Lee, Kyeong-Sik Min, and Sung-Mo (Steve) Kang. 2011. Compact Verilog-A model of phase-change RAM transient behaviors for multi-level applications. 25, 7 (2011).

[12] Dongwoo Kang, Seungjae Baek, Jongmoo Choi, Donghee Lee, Sam H Noh, and Onur Mutlu. 2015. Amnesic cache management for non-volatile memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–13.

[13] Kinam Kim and Su Jin Ahn. 2005. Reliability Investigations for Manufacturable High Density PRAM. In *IRPS*.

[14] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA*.

[15] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 2–13.

[16] Qingan Li, Lei Jiang, Youtao Zhang, Yanxiang He, and Chun Jason Xue. 2013. Compiler Directed Write-mode Selection for High Performance Low Power Volatile PCM. In *LCTES*.

[17] Qingan Li, Lei Jiang, Youtao Zhang, Yanxiang He, and Chun Jason Xue. 2013. Compiler directed write-mode selection for high performance low power volatile PCM. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 101–110.

[18] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. 2014. NVM Duet: Unified working memory and persistent store architecture. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 455–470.

[19] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*. ACM, 1.

[20] Chen Pan, Mimi Xie, Jingtong Hu, Yiran Chen, and Chengmo Yang. 2014. 3M-PCM: exploiting multiple write modes MLC phase change main memory in embedded systems. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 33.

[21] M. Poremba and Yuan Xie. 2012. NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories. In *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*. 392–397.

[22] Keni Qiu, Qingan Li, and Chun Jason Xue. 2014. Write mode aware loop tiling for high performance low power volatile PCM. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 1–6.

[23] Moinuddin K. Qureshi, Michele M. Franceschini, Luis A. Lastras-Montaño, and John P. Karidis. 2010. Morphable Memory System: A Robust Architecture for Exploiting Multi-level Phase Change Memories. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 153–162.

[24] Moinuddin K Qureshi, Michele M Franceschini, and Luis A Lastras-Montano. 2010. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–11.

[25] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 14–23.

[26] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *ISCA*.

[27] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 24–33.

[28] Guangyu Sun, Xiangyu Dong, Yuan Xie, Jian Li, and Yiran Chen. 2009. A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*. IEEE, 239–249.

[29] Cong Xu, Dimin Niu, N. Muralimanohar, R. Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. 476–488.

[30] Lunkai Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T Chong. 2016. Mellow writes: Extending lifetime in resistive memories through selective slow write backs. ISCA.

[31] Lunkai Zhang, Dmitri Strukov, Hebatallah Saadeldeen, Dongrui Fan, Mingzhe Zhang, and Diana Franklin. 2014. SpongeDirectory: Flexible Sparse Directories Utilizing Multi-level Memristors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. 61–74.

[32] Wangyuan Zhang and Tao Li. 2009. Characterizing and Mitigating the Impact of Process Variations on Phase Change based Memory Systems. In *MICRO*.

[33] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *ACM SIGARCH computer architecture news*, Vol. 37. ACM, 14–23.