Using Memory-Style Storage to Support Fault Tolerance in Data Centers

Xiao Liu Qing Yi[‡] *Jishen Zhao*
University of California at Santa Cruz, [‡]*University of Colorado Colorado Springs*
{*xiszishu,jishen.zhao*}*@ucsc.edu, qyi@uccs.edu*

Abstract

Next-generation nonvolatile memories combine byte-addressability and high performance of memory with nonvolatility of disk/flash. They promise emerging memory-style storage (MSS) systems that are directly attached to the memory bus, offering fast load/store access and data persistence in a single level of storage. MSS can be especially attractive in data centers, where fault tolerance support through storage systems is critical to performance and energy. Yet existing fault tolerance mechanisms, such as logging and checkpointing, are designed for slow block-level storage interfaces; their design choices are not wholly suitable for MSS. The goal of this work is to explore efficient fault tolerance techniques that exploit the fast memory interface and the nature of single-level storage. Our preliminary exploration shows that, by reducing data duplication and increasing application parallelism, such techniques can substantially improve system performance and energy consumption.

1 Introduction

Next-generation byte-addressable nonvolatile memories (BNVMs), such as phase-change memory, spin-transfer torque RAM, and resistive RAM, blur the boundary between memory and storage by offering both close-to-DRAM latency and nonvolatility. Attaching BNVMs directly to the processor-memory bus enables memory-style storage (MSS), which simultaneously supports data persistence (storage property) and fast load/store access (memory property). As BNVMs are anticipated to be on market in 2016 [8], MSS can radically change the performance landscape of computer systems in the near future. Yet unlocking the full potential of MSS requires substantial advancement in data manipulation schemes, which has been developed based on two-level data storage model for decades.

The need is especially acute in data centers, where storage systems are critical to reliability, availability, and energy consumption. Data centers rely on the fault tolerance capability of storage systems to ensure reliability and availability in the face of impending hardware, software, and network failures ranging from every several hours to days [5, 7]. One commonly used fault tolerance scheme combines logging and checkpointing [6, 11]. For example, Google file system (GFS) [6] maintains a critical metadata change log and checkpoints the log to local disks whenever the log grows beyond a certain size. These disk accesses drastically increases storage system energy consumption, rendering the storage system one of major energy consumers (24% of server power) in data centers servers [9].

This paper aims at exploring efficient fault tolerance with MSS in data centers. Toward this end, we investigate the performance and energy overhead of employing traditional fault tolerance techniques in MSS. Our key finding is that fault tolerance schemes, such as a combination of logging and checkpointing, can become one of the major causes of energy and performance degradation in future data center servers with MSS. We identify three major sources of performance and energy overhead, including frequent data duplication, write-order control over CPU caches, and the serialization between application operation and data persistence.

To reduce the overhead, we explore an efficient MSS fault tolerance scheme consisting of two mechanisms. The first mechanism, "interleaved persistent updates", decreases the overhead of persistent data copying through multi-versioning of the persistent updates. Second, "decoupled operation and persistence" increases the parallelism of MSS access by decoupling application operation and data persistence with separate threads. Our preliminary exploration shows that such mechanisms can effectively achieves $2.3\times$ performance improvement and $2.4\times$ energy reduction compared with traditional schemes. In particular, we make three contributions:

- Examining the performance and energy of using BNVM to support fault tolerance in data centers.
- Enhancing data center fault tolerance by exploiting the nature of MSS as a hybrid of memory and storage.
- Performing an experimental exploration of various fault tolerance mechanisms with MSS.

2 Background and Motivation

Supporting fault tolerance with MSS is challenging, because the performance and energy overhead of fault tolerance via the memory bus can overwhelm the low latency and high bandwidth offered by BNVMs. This section describes background on BNVM and data center fault tolerance techniques. We also motivate our fault tolerance design by investigating overhead sources from performing traditional data center fault tolerance schemes with MSS.

2.1 Next-generation BNVMs

BNVMs close the gap between memory and storage: they are expected to have the low access latency and high bandwidth that are close to DRAM, while offer high capacity and nonvolatility in the way as flash/disk [4, 16, 13]. Attaching BNVMs to the processor-memory bus provides a storage medium that is orders of magnitude faster than commodity hard drives and SSDs. The new paradigm inspired a large body of recent research effort on OS management, system libraries, programming models, and computer architecture. Yet most of these studies focus on how to leverage BNVMs in stand-alone computers that adopt a single type of fault tolerance techniques, e.g., logging [16], shadow paging [4], or checkpointing [14]. Substantial research is needed to explore how to best utilize BNVMs as memory-style storage (MSS) in data center environment.

2.2 Fault Tolerance in Data Centers

Most data center storage systems adopt a combination of various fault tolerance techniques, such as logging, checkpointing, and replication, to protect data from being corrupted by impending system failures [6, 11, 17]. For example, GFS [6] maintains a historical record of critical metadata changes in an operation log. Whenever the log grows beyond a certain limit, replaying them takes a considerable amount of time. To address this issue, the system checkpoints the current state using the log and dumps the checkpoint to the local disk. In case of recovery, the system loads the checkpoint into memory and replays the operation logs that are updated after the checkpoint. This work focuses on examining the problems of performing logging and checkpointing with MSS, so that we can design efficient fault tolerance techniques for data centers installed with BNVMs.

2.3 Fault Tolerance Overhead with MSS

Traditional data center fault tolerance schemes rest on two basic assumptions: 1) memory and storage are separate components so access to them are also managed separately; 2) storage is slow so the cost of the required data copying and software protocol is acceptable. However, MSS invalidates both assumptions due to its low access latency and its nature as a hybrid of memory and storage.

In the following, we investigate the performance and energy of employing traditional logging and checkpointing in this new scenario. We use a microbenchmark that repetitively swaps pairs of randomly selected elements inside a large array of strings. This benchmark maintains an in-memory log of swapping operations with a fixed number of entries; whenever the log exceeds that size, it makes a checkpoint of the array in the memory by replaying the log. During checkpointing, we accommodate subsequent swapping operations with a new log buffer. Section 4 describes our experimental setup. With our experiments, we identified three major sources of overhead.

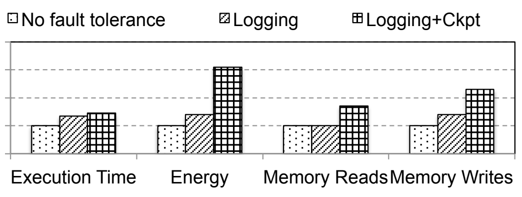
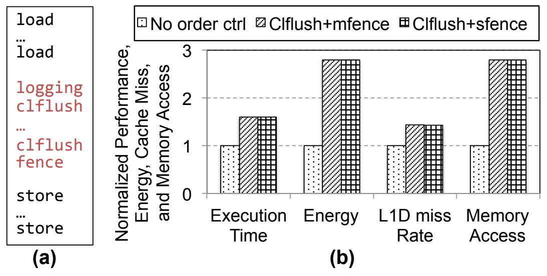


Figure 1: Normalized execution time, memory access energy consumption, and memory reads and writes of three cases: no fault tolerance support, redo logging only, and a combination of redo logging and checkpointing.



```
load
…
load
logging
clflush
…
clflush
fence
store
…
store
```

Figure 2: Overhead of write-order control. (a) Example code of a single data update, consisting of original data reads/writes, logging, cache flush, and memory fence. (b) Execution time, memory dynamic energy, cache misses, and memory access with cache flush and memory fence after issuing log updates.

1) Fine-grained Data Value Duplication. In principle, both logging and checkpointing make data copies in storage. As such, each data update yields three writes, to original data, log, and checkpoint. The load/store interface of BNVMs can accommodate logging at much finer granularity than the block I/O interface of disks/flash. For example, prior MSS design Mnemosyne [16] logs word-size data values; PMFS [1] logs 64-byte data. While the low write latency can significantly improve the efficiency of logging, the frequency of checkpointing needs to be carefully controlled to avoid high energy overhead. As shown in Figure 1, the combined logging and checkpointing scheme (*Logging+Ckpt*) introduces $3\times$ energy overhead due to significantly increased memory traffic; execution time remains similar, because we hide the checkpointing latency by accommodating subsequent logging with a new log buffer.

2) Write Order Control Over CPU Caches. To ensure the integrity of in-memory data structures, MSS needs to enforce that log updates are written into BNVM before updating the original data. Otherwise, system failures that occur when both log and original data are partially updated in BNVM lead to corrupted states in both. Such write-order control is performed by employing a combination of memory fence and CPU cache flush [16, 4] or

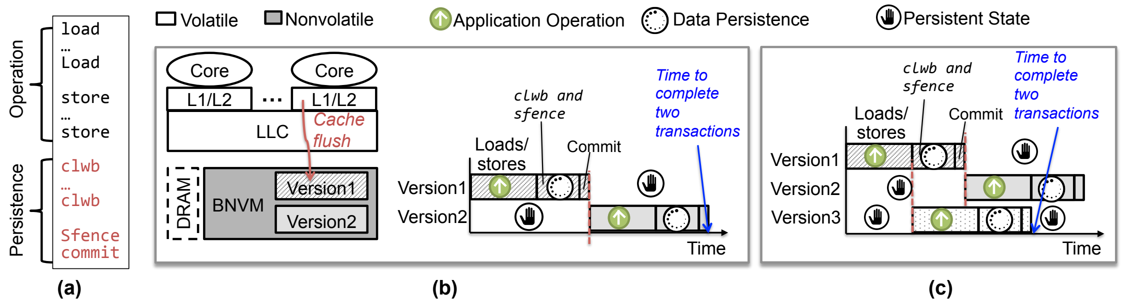


Figure 3: The interleaved persistent updates mechanism. (a) Example code of one transaction. (b) Interleaved persistent updates with two versions. (c) Optimizing interleaved persistent updates with three versions.

uncacheable writes to the log [2][1].

We add a set of `clflush` instructions, each of which flushes and invalidates one cache line out of all levels of CPU caches (Figure 2(a)). These `clflush` instructions are followed by a memory fence instruction – either `mfence` (ordering both loads and stores) or `sfence` (only ordering stores). Note that multiple `clflush` instructions implicitly order among themselves without the need of inserting memory fence in between. As expected, Figure 2(b) shows that cache flush and memory fence significantly increase memory access and energy consumption.

3) Serialization between Application Operation and Data Persistence. The write order control also reduces the parallelism that can be exploited in accessing MSS. Due to the memory fence, subsequent application operation is blocked by BNVM writes (including CPU cache flush) to log. These writes are referred as data persistence. Furthermore, with fast memory bus and fine-grained logs, MSS can perform logging at a much higher frequency than traditional disk-based storage systems. As such, frequent, serialized data persistence and application operation are one of the major reasons of performance and energy degradation shown in Figure 1 and 2.

3 MSS Fault Tolerance Mechanisms

In this section, we explore fault tolerance techniques to address the aforementioned overhead. Our design principle is two fold: 1) reducing data duplication; 2) increasing the parallelism of application operation and data persistence. To this end, we explore two MSS fault tolerance mechanisms, interleaved persistent updates and decoupled operation and persistence. The former method allows applications to directly update data without explicit logging/checkpointing or extra copying overhead. The latter exploits multithreading to enable concurrency among normal application operations and data persistence operations.

3.1 Interleaved Persistent Updates

With logging and checkpointing, log buffer and checkpointing area are needed in addition to original data. The same data values are copied among these memory regions. Instead, we explore ways to support fault tolerance by directly making each write persistent without copying.

Key Idea. Figure 3(a) and (b) illustrate our key idea of this mechanism[2]. Each *transaction* (a group of instructions that atomically perform MSS updates [16, 3]) consists of reads and writes of the original data, cache flushes, and a memory fence, however, no logging or checkpointing. Fault tolerance is supported by multiversioning. As demonstrated in Figure 3(b), we allocate two versions of data in memory. Each application operation directly overwrites one version, referred as an *operating version*; the other version, the *persistent version*, remains intact and can be used to recover MSS. After all writes to the operating version reach BNVM, we commit these updates and turn them to a checkpoint. We assign a bit in each version to indicate such committing. When a future transaction needs to update the same data, we switch the operating and persistent versions. As such, we interleave data updates between the two versions without writing the same data value multiple times.

Performance Optimization. With only two versions, consecutive transactions that update the same data can be serialized (Figure 3(b)). To optimize performance, we develop a three-version design as illustrated in Figure 3(c). As long as one version stays in persistent state (e.g., Version2 in the figure), we can simultaneously perform application operation and data persistence in two other versions (e.g., data persistence in Version1 and application operation in Version3). Increasing the number of versions can further improve parallelism. But doing so will increase storage overhead and implementation complexity. Therefore, we do not evaluate designs with more versions.

[1]Another method is write-through caching, but is less desirable due to inflexibility and performance issues [2, 16, 4].

[2]while servers may employ hybrid DRAM/BNVM memory, our work focuses on supporting fault tolerance using BNVM.

Algorithm 1: Pseudo-code of the operation thread.

```
Initialization and version setup;
for each data update do
    Issue $update_i$;
    for $store_j \in update_i$ do
        $store_j$;
    end
    if (Persistence==done) and (Other conditions) then
        Persistence=doing;
        Reset intermediate value;
    end
end
```

Algorithm 2: Pseudo-code of the persistence thread.

```
while loads and stores are not completed do wait;
for every updated element in the persistent data structure do
    clwb(element.memaddr);
    Version control operations;
end
sfence;
Mark the version as persistent;
Set Persistence=done;
```

3.2 Decoupled Operation and Persistence

Most previous work performs original data updates, cache flushes, memory fence, and commits in a single thread. We exploit parallelism within each transaction by decoupling application operation and data persistence. To achieve this, we employ two threads, an operation thread to issue data update requests and a persistence thread to perform cache flush, memory fence, and commits (Algorithm 1 and 2). The decoupling allows us to avoid enforcing data to be persistent immediately after each memory operation by the application. Instead, when the persistence thread is idle, it starts to persist the recently updated data. The two threads are synchronized to avoid flushing data that is still being updated. The underline rationale is that the fast memory interface of MSS can accommodate much more frequent persistent data updates than needed.

We also allow users to manually tune the data persistence frequency by adding "*other conditions*" in Algorithm 1. Section 4 shows the sensitivity of performance and energy with respect to such frequency.

4 Preliminary Exploration

We measure the performance and memory access statistics of our workload on a machine configured with eight Intel Corei7-4790 cores running at 3.6 GHz. Each core has 32KB L1 data cache and share 8MB last-level cache. The machine has two 8GB DDR3-1600 DRAMs. We use `perf`, a Linux profiling tool to collect cache and memory access statistics. The access latency of BNVM is typically longer than DRAM. Therefore, we can use statistical emulation methods similar to previous studies [15] to emulate BNVM. We use McPAT [10] to calculate processor power. We model the dynamic energy consumption of spin-transfer torque memory based on the number of memory accesses: each row buffer hit consumes 0.93 (1.02) pJ/bit, each memory array read (write) consumes 1.00 (2.89) pJ/bit [18]. We use Pin [12] to profile the row buffer hit rate of the workload.

Our experiments are performed on the SPS benchmark [3], which randomly swaps two elements in a large array of strings. We randomly generate 100,000 of 256-byte strings in the array and report results with 100K and 1M swaps. Intel's next generation cache flush instruction `clwb`, which flushes a cache line without invalidating it, offers better performance than `clflush`. However, we employ `clflush` instead because `clwb` is not publicly available. We evaluate the following eight implementations of the SPS benchmark:

- **DiskLogging**(*DL*): Performs redo logging (write new data values in the log) at the end of each transaction and periodically writes the log as a disk file.
- **Logging**(*L*): Performs redo logging in BNVM without CPU cache flush or memory fence, i.e., without data persistence.
- **Logging+clflush+mfence**(*LCM*): Enforces data persistence using a set of `clflush` instructions and a `mfence`.
- **Interleaved+clflush+mfence**(*ICM*): Performs interleaved persistent updates instead of logging, with `clflush` and `mfence` to enforce data persistence.
- **Threading+clflush+mfence**(*TCM*): Performs interleaved persistent updates and decoupled operation and persistence, with `clflush` and `mfence` to enforce data persistence.
- **Threading+clflush+sfence**(*TCS*): Uses `sfence` instead of `mfence` in the previous implementation.
- **Logging+clflush+sfence**(*LCS*), **Interleaved + clflush + sfence**(*ICS*), and **Threading+clflush+sfence**(*TCS*): Uses `sfence` instead of `mfence` in corresponding mechanisms.

We run each instance of the benchmark for 200 times and calculate the average performance, energy consumption, and memory access. The variations over different runs are low ($<0.1\%$) so they are not shown in our results. Our evaluation does not add checkpointing on top of logging, although doing so can demonstrate more benefits of our methods.

4.1 Performance

Our results show that *Logging* leads to dramatic performance and energy improvements compared to *DiskLogging*: 23.7× operation throughput improvement and 16.1× energy reduction. Consequently, we use *Log-*

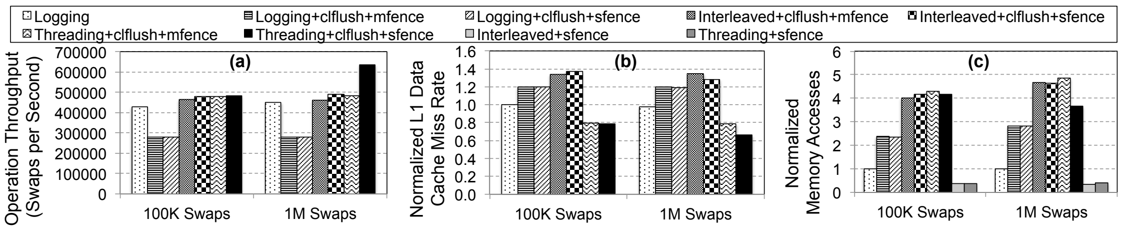


Figure 4: Operation throughput, L1 data cache miss rate, and memory access of various SPS implementations.

ging as the baseline in the rest of this section.

Figure 4(a) shows the swap operation throughput across various implementations, measured as the number of swaps completed per second. We make three major observations. First, cache flush and memory fence degrade system performance by more than 1.6× compared with logging without enforcing data persistence. Using `sfence` does not improve performance, because the CPU cache flushes, which are implicitly ordered by memory fences, dominate the performance overhead. Second, our two MSS fault tolerance mechanisms substantially improve system performance compared to logging with data persistence enforced, due to reduced memory traffic. Third, performance of *TCS* scales much better than others, when we increase the number of swaps. It achieves 2.3× throughput of logging mechanisms. This is because the larger number of swap operations offers better opportunity for our mechanisms to exploit parallelism between operation and data persistence.

4.2 Memory Traffic

Our memory traffic results align with the performance results. We make two observations from the L1 cache miss rates of various mechanisms (normalized to *Logging*, Figure 4(b)). First, the interleaved persistent updates mechanism increases L1 cache miss rate by 10% compared with *LCS*. This is because `clflush` forces original data updates, which can be re-accessed later on during application operation, out of the cache. Yet, *TCM* and *TCS* reduce L1 cache miss rate by 53% compared with *LCS* because cache is less frequently flushed by decoupling the swap operations and data persistence.

Figure 4(c) demonstrates the number of BNVM access normalized to *Logging*. *LCM* and *LCS* result in more than doubled memory access compared with the baseline, due to the cache flush of the duplicated data values written in the log. Our mechanisms further increase the number of memory access. This is because `clflush` invalidates the flushed cache lines and leads to additional cache misses from subsequent accesses to these cache lines. We investigate the overhead of the invalidation by removing the `clflush` instructions in the benchmark. As a result, *IM* and *TS* lead to dramatically reduced memory access. Note that without cache flushes, we still preserves data persistence by using memory fence with an increase of latency.

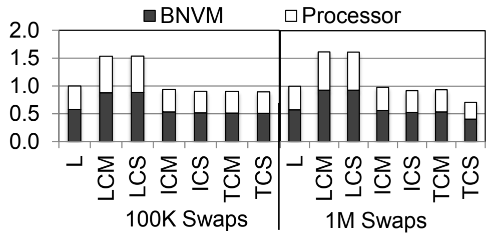


Figure 5: Energy consumption breakdown of various SPS implementations.

4.3 Energy Consumption

Figure 5 shows the dynamic energy breakdown of various mechanisms normalized to *Logging*. First of all, BNVM access consumes a large portion of system energy, due to large number of memory access performed during data persistence. Cache flush and memory fence further increase the memory dynamic energy consumption due to increased number of memory access. It appears surprising that *ICM* and *ICS* consume much lower memory dynamic energy than *LCS* with similar numbers of memory access. But taking a closer look at the memory access, we observe that most of the increased memory access are reads that consume much less energy than BNVM writes in the interleaved mechanisms. Overall, our MSS fault tolerance mechanisms lead to 2.4× energy reduction compared with *LCS*. While not quantitatively demonstrated, our mechanisms can save even more energy compared with traditional fault tolerance schemes that perform both logging and checkpointing.

4.4 Sensitivity to Persistence Frequency

Persistence frequency can determine the performance and energy of *TCM* and *TCS*. We investigate such relationship by sweeping across various persistence thresholds, which is defined as the ratio of data persistence operations to application operations. Figure 6 and 7 shows system dynamic energy consumption normalized to those with a persistent threshold of 0.2. It shows that we can achieve minimum energy consumption and high swap operation throughput at the threshold of 0.6. Further increasing the threshold, i.e., higher data persistence rates, can increase the total number of memory access due to the frequent cache flushes. Lower data persistence frequency leads to lower fault tolerance. Therefore, we employed threshold 0.6 across our performance, memory access, and energy evaluations. Different applica-

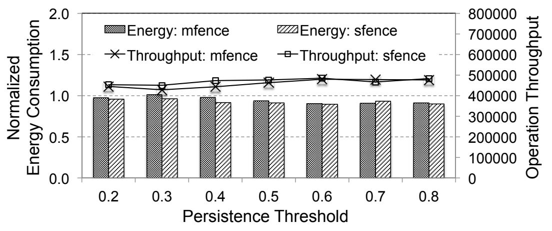


Figure 6: Sensitivity of performance and energy to the persistence frequency of the persistence thread under 100k swaps.

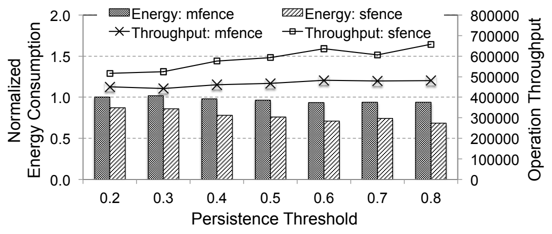


Figure 7: The same metric as Figure 6 under 1M swaps.

tions may favor different thresholds. We allow users to set the threshold or determine it by application profiling.

5 Discussion

This paper performs a preliminary exploration toward unleashing the full potential of MSS in data centers. We foresee several open research questions that arise when supporting efficient fault tolerance with MSS. First, while we evaluated our scheme with an array data structure, the key ideas are also applicable to various other data structures (e.g., linked list, hash table, B+tree, and graph) used in storage systems. To fully achieve the potential energy and performance benefits of MSS, we would like to tailor and evaluate the scheme for various data structures. Second, in addition to logging and checkpointing, data centers also employ replication [17] to support fault tolerance. We would like to investigate how our scheme interact with replication mechanism. Third, whereas our design does not require logging to ensure fault tolerance, logs can offer the portability of data being used across incompatible system configurations. Therefore, we would like to explore efficient mechanisms for logging that serves such functions.

6 Conclusions

We have explored an efficient scheme that uses memory-style storage to support fault tolerance in data centers. Our scheme consists of two mechanisms, to reduce memory traffic in logging/checkpointing and increase the parallelism between application operation and data persistence. Our results demonstrate the support of fault tolerance with low cost by leveraging the nature of MSS as a hybrid of memory and storage, rather than simply manipulating it as either. As such, our exploration paves the way for unlocking the full potential of BNVMs in data center storage systems.

7 Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by UCSC startup funding, NSF I/UCRC Center for Research in Storage Systems at UCSC, NSF awards CCF1261811, CCF1421443, and CCF1261584.

References

[1] Intel, Persistent memory file system, https://github.com/linux-pmfs/pmfs.

[2] BHANDARI, K., CHAKRABARTI, D. R., AND BOEHM, H.-J. Implications of CPU caching on byte-addressable non-volatile memory programming. In *HPL Tech. Repo.* (2012), pp. 1–6.

[3] COBURN, J., CAULFIELD, A. M., AKEL, A., AND ET AL. NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS* (2011), pp. 105–118.

[4] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *SOSP* (New York, NY, USA, 2009), pp. 133–146.

[5] DI, S., ROBERT, Y., VIVIEN, F., KONDO, D., WANG, C.-L., AND CAPPELLO, F. Optimization of cloud task processing with checkpoint-restart mechanism. In *SC* (2013), pp. 64:1–64:12.

[6] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[7] HUANG, S., FU, S., ZHANG, Q., AND SHI, W. Characterizing disk failures with quantified disk degradation signatures: An early experience. In *IISWC'15* (2015), pp. 1–10.

[8] INTEL, AND MICRON. Intel and Micron produce breakthrough memory technology, 2015. http://newsroom.intel.com/.

[9] KOOMEY, J. Growth in data center electricity use 2005 to 2010. In *A report by Analytical Press, completed at the request of The New York Times* (2011), pp. 1–24.

[10] LI, S., AHN, J. H., STRONG, R. D., BROCKMAN, J. B., TULLSEN, D. M., AND JOUPPI, N. P. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO* (2009), pp. 469–480.

[11] LINKEDIN. Running Kafka at scale, 2015.

[12] LUK, C.-K., COHN, R., MUTH, R., AND ET AL. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI* (New York, NY, USA, 2005), pp. 190–200.

[13] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *ISCA* (2014), pp. 1–12.

[14] REN, J., ZHAO, J., KHAN, S., CHOI, J., WU, Y., AND MUTLU, O. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *MICRO* (2015), pp. 1–13.

[15] SENGUPTA, D., WANG, Q., VOLOS, H., AND ET AL. A framework for emulating non-volatile memory systemswith different performance characteristics. In *ICPE* (2015), pp. 317–320.

[16] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *ASPLOS* (2011), pp. 91–104.

[17] ZHANG, Y., YANG, J., MEMARIPOUR, A., AND SWANSON, S. Mojim: A reliable and highly-available non-volatile memory system. In *ASPLOS* (2015), pp. 3–18.

[18] ZHAO, J., LI, S., YOON, D. H., XIE, Y., AND JOUPPI, N. P. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO* (2013), pp. 421–432.