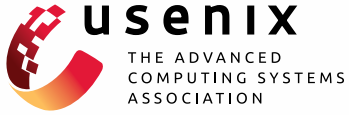


Request-Oriented Durable Write Caching for Application Performance

Sangwook Kim, ***Sungkyunkwan University;*** **Hwanju Kim,** ***University of Cambridge;***
Sang-Hoon Kim, ***Korea Advanced Institute of Science and Technology (KAIST);***
Joonwon Lee and Jinkyu Jeong, ***Sungkyunkwan University***

https://www.usenix.org/conference/atc15/technical-session/presentation/kim

This paper is included in the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15) is sponsored by USENIX.

Request-Oriented Durable Write Caching for Application Performance

Sangwook Kim[†], Hwanju Kim[§,*], Sang-Hoon Kim[‡], Joonwon Lee[†], Jinkyu Jeong[†]
[†]*Sungkyunkwan University,* [§]*University of Cambridge,* [‡]*KAIST*
{*sw.kim, joonwon, jinkyu*}*@skku.edu, hwandori@gmail.com, sanghoon@calab.kaist.ac.kr*

Abstract

Non-volatile write cache (NVWC) can help to improve the performance of I/O-intensive tasks, especially write-dominated tasks. The benefit of NVWC, however, cannot be fully exploited if an admission policy blindly caches all writes without differentiating the criticality of each write in terms of application performance. We propose a request-oriented admission policy, which caches only writes awaited in the context of request execution. To accurately detect such writes, a critical process, which is involved in handling requests, is identified by application-level hints. Then, we devise criticality inheritance protocols in order to handle process and I/O dependencies to a critical process. The proposed scheme is implemented on the Linux kernel and is evaluated with PostgreSQL relational database and Redis NoSQL store. The evaluation results show that our scheme outperforms the policy that blindly caches all writes by up to 2.2× while reducing write traffic to NVWC by up to 87%.

1 Introduction

For decades, processor and memory technologies have been significantly improved in terms of performance whereas the performance of storage still lags far behind that of other components. To remedy this performance gap, modern operating systems (OSes) use main memory as a cache for underlying storage. With large main memory, this technique is effective for read-intensive tasks by hiding long latency of storage reads [63]. For write operations, however, caching is less effective because the volatility of main memory may lead to data loss in the event of power failure. As a consequence, write operations dominate the traffic to storage in production workloads operating with large main memory [13, 50, 69, 77].

Non-volatile write cache (NVWC) can help to improve the performance of I/O-intensive tasks, especially write-dominated tasks. For this reason, battery-backed DRAM (NV-DRAM) has been widely exploited as an NVWC device for file systems [11, 15, 41], transaction processing systems [27, 57, 74], and disk arrays [36, 37]. In addition, various caching solutions based on flash memory have been extensively studied to efficiently utilize fast random access of flash memory [10, 19, 46, 49, 65]. Storage-class memory (SCM), such as spin-transfer torque magneto-resistive memory (STT-MRAM) [17] and phase change memory (PCM) [67], is expected to be deployed as NVWC since it provides low latency comparable to DRAM and persistency without backup battery.

Blindly caching all writes, however, cannot fully utilize the benefit of NVWC for application performance due to the following reasons. Firstly, it can frequently stall writes in the performance-critical paths of an application due to the lack of free blocks in NVWC, especially for capacity-constrained devices, such as NV-DRAM and STT-MRAM. Secondly, it can cause severe congestion in OS- and device-level queues of NVWC, thereby delaying the processing of performance-critical writes. Finally, it would hurt the reliability and performance depending on the characteristics of the NVWC device used. For instance, caching non-performance-critical writes exacerbates wear-out of storage medium, such as flash memory [39, 40, 80] and PCM [17, 66], without any gain in application performance.

We propose a *request-oriented admission policy* that only allows *critical writes* (i.e., performance-critical writes) to be cached in NVWC. In particular, we define critical writes as the writes awaited in the context of *request execution* since the performance of processing an external *request*, like a key-value PUT/GET, determines the level of application performance. By using the proposed policy, a large amount of non-critical writes can be directly routed to backing storage bypassing NVWC because typical data-intensive applications, such as relational database management system (RDBMS) [38, 59] and NoSQL store [25, 31], delegate costly write I/Os

[*]Currently at EMC

to background processes while concurrently handling requests using other processes; we refer to any kind of execution context as process in this paper.

The key challenge of realizing the proposed policy is how to accurately identify all critical writes. Basically, synchronous writes requested by a *critical process*, which is involved to handle requests, are critical writes by definition. This simple identification, however, cannot detect process and I/O dependency-induced critical writes generated by complex synchronizations during runtime. Since synchronization is the technique frequently used to ensure correct execution among concurrent processes and I/Os, unresolved dependencies can significantly delay the progress of a critical process, thereby degrading application performance.

We devise *hint-based critical process identification* and *criticality inheritance protocols* for accurate detection of critical writes. Basically, the proposed scheme is guided by a hint on a critical process from an application. Based on the given hint, synchronous writes requested by a critical process are cached in NVWC. To handle process dependency, we inherit criticality to a non-critical process on which a critical process depends to make progress. To handle I/O dependency, we dynamically reissue an outstanding non-critical write with which a critical process synchronizes to NVWC without compromising the correctness. We also resolve cascading dependencies by tracking blocking objects recorded in the descriptors of processes who have dependencies to a critical process.

Our proposed scheme was implemented on the Linux kernel and FlashCache [2]. Based on the prototype implementation, we evaluated our scheme using PostgreSQL [5] and Redis [23] with a TPC-C [7] and YCSB [26] benchmark, respectively. The evaluation results have shown that the proposed scheme outperforms the policy that blindly caches all writes by 3–120% and 17–55% while reducing write traffic to NVWC by up to 72% and 87%, for PostgreSQL and Redis, respectively.

Our key contributions are the followings:

- We introduce a novel NVWC admission policy based on request-oriented write classification.
- We devise criticality inheritance protocols to handle complex dependencies generated during runtime.
- We prove the effectiveness of our scheme by conducting case studies on real-world applications.

The remainder of this paper is organized as follows: Section 2 describes the background and motivation behind this work. Section 3 and Section 4 detail the design of the proposed policy. Section 5 explains the prototype implementation, and Section 6 presents our application studies. Section 7 presents the evaluation results. Finally, Section 8 presents related work and Section 9 concludes our work and presents future direction.

2 Background and Motivation

2.1 Non-volatile Write Caches

Unlike conventional volatile caches, non-volatile write cache (NVWC) is mainly used to durably buffer write I/Os for improving write performance. Traditionally, NV-DRAM has been widely used as an NVWC device to enhance write performance by exploiting its low latency and persistency. Typical usages of NV-DRAM are writeback cache in RAID controllers [36, 37] and drop-in replacement for DDR3 DIMMs [8, 78]. An inherent limitation of NV-DRAM is small capacity due to high cost per capacity and battery scaling problem.

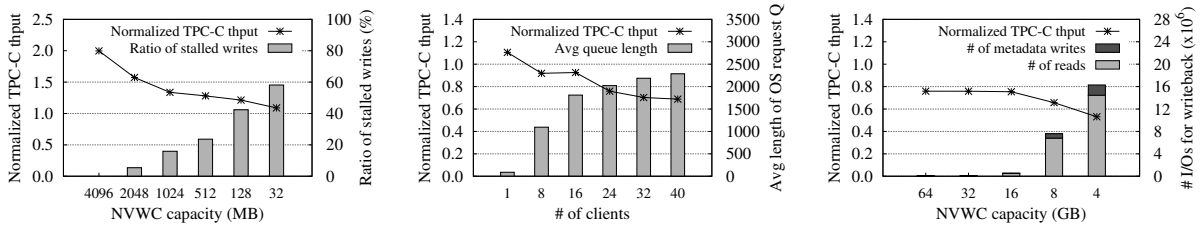
Recently, flash memory-based caching is gaining significant attention because it delivers much higher performance than traditional disks and much higher density than NV-DRAM. Thus, flash memory is widely adopted in many storage solutions, such as hybrid storage [19, 70] and client-side writeback caches in networked storage systems [10, 49, 65]. Despite of the benefits, flash memory also has caveats to be used as an NVWC device because it has limited write endurance [39, 40, 80] and garbage collection overheads [44, 45, 47, 62].

Emerging SCM, such as STT-MRAM [17] and PCM [67], is also a good candidate for an NVWC device since it provides low latency comparable to DRAM and persistency without backup power. Though STT-MRAM promises similar access latency to that of DRAM, its capacity is currently very limited due to technical limitation [1]. On the other hand, PCM has been regarded as more promising technology to be deployed at commercial scale than STT-MRAM [3]. PCM, however, has limited write endurance [52, 66], which necessitates careful management when it is used as an NVWC device.

2.2 Why Admission Policy Matters

A straightforward use of NVWC is to cache all writes and to writeback cached data to backing storage in a lazy manner. This simple admission policy is intended to provide low latency for all incoming writes as much as possible for improving *system performance* (e.g., IOPS). However, blindly caching all writes cannot fully utilize the benefit of NVWC in terms of *application performance* (e.g., transactions/sec) for the following reasons.

Firstly, caching all writes can frequently stall writes that are in the critical paths of an application due to the lack of free blocks in NVWC. This is because the speed of making free blocks is eventually bounded by the writeback throughput to backing storage, such as



(a) Stalled writes (ramdisk) (b) Queueing delay (64GB SSD) (c) Concurrent I/Os for writeback (SSD)

Figure 1: **Limitations of caching all writes.** *TPC-C throughput is normalized to the case of No NVWC.*

disks. The stalled writes problem becomes more serious in capacity-constrained devices, such as NV-DRAM and STT-MRAM. In order to quantify the impact of stalled writes on application performance, we ran a TPC-C benchmark [7] against PostgreSQL RDBMS [5] on a Linux-based system having NV-DRAM-based NVWC (emulated via ramdisk); see Section 7 for the detailed configuration. As shown in Figure 1(a), the TPC-C throughput normalized to the case without NVWC (i.e., disk only) drops from 1.99 to 1.09 as the capacity of NVWC decreases. This is because the frequency of write stalls in critical paths of PostgreSQL (e.g., stalls during log commit) is highly likely to increase as the ratio of stalled writes increases.

Secondly, caching all writes can incur significant congestion in OS- and device-level request queues of NVWC, thereby delaying the processing of critical writes. When the request queues of NVWC are congested, write requests need to wait at the queues even though the NVWC has sufficient free blocks. Moreover, queue congestion of a storage-based NVWC such as SSD can be exacerbated by concurrent I/Os for writing back cached data. The concurrent I/Os include NVWC reads for retrieving a dirty cache block into main memory and NVWC writes for updating the corresponding metadata. In order to measure the impact of aggravated queueing delay, we ran the TPC-C benchmark with a flash SSD-based NVWC. As shown in Figure 1(b), the average length of OS request queue increases as the number of clients increases, thereby gradually degrading the normalized TPC-C throughput. In addition, the performance further decreases as the concurrent I/Os increase as shown in Figure 1(c); the frequency of writebacks depends on the ratio of dirty blocks in NVWC for this measurement. In most cases, NVWC provides even lower performance than that without NVWC (up to 47% performance loss) though write stalls did not occur at all in all the configurations.

Finally, caching all writes would hurt reliability and performance depending on the characteristics of an NVWC device. For example, caching non-critical writes exacerbates the wear-out of an NVWC device, like flash and PCM, without any gain in application performance. In addition, caching non-critical writes can increase the probability of garbage collection while processing critical writes in flash-based NVWCs.

For these reasons, caching only critical writes to NVWC is vital to fully utilize a given NVWC device for application performance. From the analysis based on the realistic workload (Section 7.2), we found that all writes do not equally contribute to the application performance. This finding implies that there is a need to classify write I/Os for typical data-intensive applications such as databases and key-value stores.

3 Which Type of Write is Critical?

3.1 Request-Oriented Write Classification

The primary role of a data-intensive application is to provide a specific data service in response to an external *request*, like a PUT/GET request to a key-value store. In such an application, the performance of request processing determines the level of application performance a user perceives. Therefore, we need to identify which type of writes delays the progress of request processing to classify critical writes.

Synchronous writes can be a good candidate for the type of critical writes. Traditionally, write I/O is broadly classified into two categories in the system's viewpoint: asynchronous and synchronous. When a process issues an asynchronous write, it can immediately continue processing other jobs without waiting for the completion of the write. A synchronous write, on the other hand, is awaited by a requesting process until the write completes. Due to this difference, prioritizing synchronous writes over asynchronous ones is known as a reasonable method to reduce system-wide I/O wait time [35], and hence it is adopted in commodity OS [28].

However, not all synchronous writes are truly synchronous from the perspective of *request execution*. Typical data-intensive applications delegate a large amount of synchronous writes to a set of background processes as a way of carrying out internal activities. For instance, RDBMS [38, 59] and NoSQL store [25, 31] adopt a variant of logging technique that accompanies only a small amount of (mostly sequential) synchronous writes during request processing while conducting a burst of (mostly

Write Type	Process	Ratio (%)
Sync.	backends	44.312
	checkpointer	34.664
	log writer	0.368
	jbd2 (kernel)	0.094
	etc	0.007
Async.	kworker (kernel)	20.554
	etc	0.002

Table 1: **Breakdown of writes by type and process.**

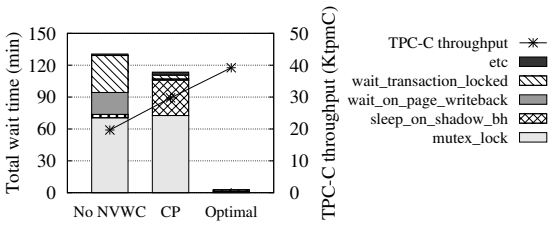


Figure 2: **Impact of dependencies on application performance.** *CP caches synchronous writes requested by critical processes while Optimal caches all writes without stalls. Network latencies are omitted for brevity.*

random) synchronous writes in background. This is an intrinsic design to achieve high degree of application performance without loss of durability by segregating costly synchronous writes from the critical path of request execution as much as possible.

To verify such behaviors, we ran the TPC-C benchmark using 24 clients without NVWC and recorded the type of write issued per process. As shown in Table 1, about 80% of the writes are synchronous, and most of them are performed by *backends* and *checkpointer*. In PostgreSQL, the backend is a dedicated process for handling requests while the checkpointer periodically issues a burst of synchronous writes to reflect buffer modifications to backing storage. Likewise, in kernel-level, journaling daemon (i.e., *jbd2*) also issues synchronous writes (though small amount in this case) for committing and checkpointing file system transactions. Basically, the synchronous writes requested by the processes other than the backends are irrelevant to request processing. Furthermore, according to our analysis result (Table 3), asynchronous writes occasionally block the backends because of complex synchronizations during runtime. The conventional synchrony-based classification, therefore, is inadequate for classifying critical writes.

We introduce *request-oriented write classification* that classifies a write awaited in the context of request execution as a critical write regardless of whether it is issued synchronously or not. Based on this classification, only critical writes are cached into NVWC while non-critical writes are routed to backing storage directly. As a result, a request can be handled quickly by avoiding excessive write stalls and queue congestion. In addition, device-specific reliability and performance issues, which are discussed in Section 2.2, can be eased without hurting application performance.

3.2 Dependency-Induced Critical Write

In data-intensive applications, one or more processes are involved in handling requests. Synchronous writes issued by these processes are definitely critical; hence, we refer to this type of processes as a *critical process*. Caching these synchronous writes alone, however, is insufficient for identifying all critical writes. This is because runtime dependencies generated by complex synchronizations among concurrent processes and I/Os make critical processes wait for the writes that are not synchronously issued by them.

There are two types of dependencies associated with write I/Os: a *process dependency* and an *I/O dependency*. The process dependency occurs when two processes interact with each other via synchronization primitives, such as a lock and condition variable. The process dependency complicates the accurate detection of critical writes because a non-critical process may issue synchronous writes within a critical section making a critical process indirectly wait for the completion of the writes. On the other hand, the I/O dependency occurs between a critical process and an ongoing write I/O. Basically, the I/O dependency is generated when a critical process needs to directly wait for the completion of an outstanding write in order to ensure consistency and/or durability.

In order to quantify the significance of the dependency problems, we measured the wait time of critical processes (i.e., PostgreSQL backends) using *LatencyTOP* [34] during the execution of the TPC-C benchmark using 24 clients with 4GB ramdisk-based NVWC. Figure 2 shows the impact of complex dependencies on the TPC-C throughput; CP caches only synchronous writes requested by critical processes while Optimal caches all writes without stalls and queue congestion. As we expect, CP mostly eliminates the latency incurred by synchronous writes (i.e., `wait_on_page_writeback()`). However, CP still suffers from excessive latencies mainly caused by process dependency (i.e., `mutex_lock()`) and I/O dependency (i.e., `sleep_on_shadow_bh()`). Note that the I/O dependency occurs because a critical process attempts to update a buffer page that is under writing back as the part of a committing file system transaction. Consequently, CP achieves only a half of the performance improvement compared to Optimal.

In addition, there are many other sources of excessive latencies in terms of the average and worst case as shown in Table 2. The read/write semaphore for serializing on-disk inode modifications represented as `down_read()` induces about one second and several seconds latencies in the average and worst case, respec-

tively. The journaling-related synchronizations for ensuring file system consistency also incur high latencies. In particular, `wait_transaction_locked()` is called to synchronize with all the processes updating the current file system transaction to complete their execution while `jbd2_log_wait_commit()` is called to wait for the journaling daemon to complete the commit procedure of a file system transaction. The synchronization methods that induce I/O dependency, such as `lock_buffer()` and `lock_page()`, delay the progress of critical processes up to several seconds. Though some of the synchronization methods account for a small portion of the total latency, they would increase tail latency, thereby degrading user experience in large-scale services [29]. Therefore, all synchronization methods causing latency to the critical processes should be handled properly in order to eliminate unexpected request latency spikes.

4 Critical Write Detection

4.1 Critical Process Identification

In order to detect all critical writes, we should identify critical processes in the first place. To do so, we adopt an application-guided approach that exploits application-level hints.

The main benefit of the application-guided approach is that it does not increase the complexity of the OS kernel. Accurately identifying critical processes without application guidance requires huge engineering effort to the kernel. For instance, similar to the previous approaches [83, 84], the kernel should track all inter-process communications and network-related I/Os to infer the processes handling requests. In addition, the kernel should adopt complex heuristics (e.g., feedback-based confidence evaluation [84]) to reduce the possibility of misidentification. On the other hand, an application can accurately decide the criticality of each process since the application knows the best which processes are currently involved in handling requests.

Though the application-guided approach requires application modifications, the engineering cost for the modifications is low in practice. This is because an application developer does not need to know the specifics of underlying systems since the hint (i.e., disclosure [64]) revealing a critical process remains correct even when the execution environment changes. In addition, typical data-intensive applications, such as MySQL [4], PostgreSQL, and Redis, already distinguish foreground processes (i.e., critical processes) from background processes. This distinction is also common for event-driven applications since they need to clearly separate internal activities from request flows as exemplified in Cassandra [43]. As a consequence, the required modification is only a few lines of code in practice; see Section 6 for our application studies.

Dep. Type	Synchronization Method	Avg (ms)	Max (ms)
Process	`down_read`	1088.09	6065.2
	`wait_transaction_locked`	493.05	4806.8
	`mutex_lock`	134.55	6313.55
	`jbd2_log_wait_commit`	40.96	391.36
I/O	`lock_buffer`	912.38	3811.35
	`sleep_on_shadow_bh`	225.25	3560.47
	`lock_page`	8.08	3009.84
	`wait_on_page_writeback`	0.04	19.12

Table 2: **Sources of dependencies.** *Average and maximum wait times of backends are shown in the CP case.*

Since a hint is solely used for deciding admission to NVWC, a wrong hint does not affect the correct execution of an application. However, hint abuse by a malicious or a thoughtless application may compromise performance isolation among multiple applications sharing NVWC. This problem can be solved by overriding criticality of each write at the kernel based on a predefined isolation policy. Addressing the issue resulting from sharing NVWC is out of scope of this paper.

4.2 Criticality Inheritance Protocols

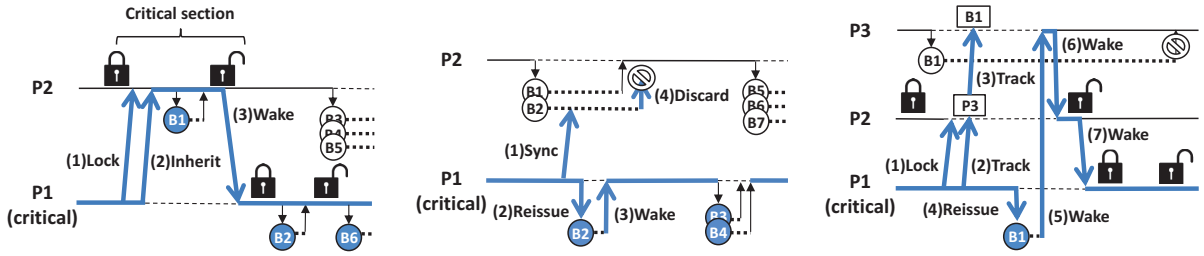
As we discussed in Section 3.2, the process and I/O dependencies can significantly delay the progress of a critical process. In the rest of this section, we explain our *criticality inheritance protocols* that effectively resolve the process and I/O dependencies.

4.2.1 Process Criticality Inheritance

Handling the process dependency has been well-studied in the context of process scheduling because the process dependency may cause priority inversion problem [51]. Priority inheritance [72] is the well-known solution for resolving the priority inversion problem.

Inspired by the previous work, we introduce *process criticality inheritance* to resolve the process dependency. Process criticality inheritance is similar to the priority inheritance in that a non-critical process inherits criticality when it blocks a critical process until it finishes its execution within the synchronized region. The main difference between process criticality inheritance and priority inheritance is that the former is used to prioritize I/Os whereas the latter is used to prioritize processes.

Figure 3(a) illustrates an example of process criticality inheritance: (1) critical process P1 attempts to acquire a lock to enter a critical section. (2) Non-critical process P2 inherits criticality from P1 since the lock is held by P2. Then, the synchronous write to block B1 issued by P2 is directed to NVWC to accelerate the write within



(a) Process criticality inheritance (b) I/O criticality inheritance (c) Handling cascading dependencies

Figure 3: **Criticality inheritance protocols.** *Thick lines represent a critical path of request execution while dotted lines indicate blocked execution. Circles and boxes represent write I/Os to specific blocks and blocking objects of specific processes, respectively. Thick arrows indicate specific actions described in the corresponding texts while thin arrows to/from write I/Os show I/O submission/completion.*

the critical path. (3) P2 wakes up P1 when its execution within the critical section has been completed, and P1 continues the rest of its job. In this example, the write latency of block B1 is minimized by using process criticality inheritance since it indirectly delays the request execution.

4.2.2 I/O Criticality Inheritance

Handling the I/O dependency is more complicated than that of the process dependency since inheriting criticality to the ongoing write to backing storage requires reissuing the write to NVWC without side effects (e.g., duplicated I/O completion). A possible solution for eliminating the side effects is canceling an outstanding write to a disk. In practice, however, canceling a specific ongoing write needs significant engineering efforts due to multiple abstraction stages in I/O stack. In Linux, for example, an I/O can be staged in either an OS queue managed by an I/O scheduler or a storage queue managed by a device firmware. Hence, the procedure of canceling requires lots of modifications to various in-kernel components.

In order to rapidly resolve the I/O dependency while maintaining low engineering cost, we devise immediate reissuing and I/O completion discarding as a technique for I/O criticality inheritance. Figure 3(b) describes the proposed mechanism: (1) critical process P1 needs to wait for the completion of the write request to block B2. (2) P1 reissues B2 to NVWC to resolve the I/O dependency between P1 and B2. (3) The event of I/O completion of the reissued B2 wakes up P1. (4) Later, the I/O completion of the original write to B2 is discarded to suppress the duplicated completion notification.

The main drawback of the proposed technique for I/O criticality inheritance is that it cannot eliminate unnecessary write traffic to a disk since it does not cancel ongoing writes to the disk. However, the performance penalty would be small since the duplicated write to a specific block is highly likely to be processed as a single sequential write merged with other writes to adjacent blocks in modern OSes. Moreover, discarding several blocks included in a write request may result in splitting the request into multiple requests, thereby decreasing the efficiency of I/O processing. In practice, the amount of reissued writes is insignificant despite of its large contribution to application performance (Table 3).

4.2.3 Handling Cascading Dependencies

Cascading dependencies, a chain of process and I/O dependencies, make precise detection of critical writes more difficult if the chain contains a process that is already blocked. For example, as illustrated in Figure 3(c), non-critical process P3 issues a synchronous write and is blocked to wait for the completion of the write. Later, non-critical process P2 sleeps while holding a lock because it needs to wait for an event from P3. In this situation, if critical process P1 attempts to acquire the lock that is held by P2, P1 blocks until the write issued by P3 is completed even though P2 inherits criticality from P1. We found that this scenario occurs in practice because of complex synchronization behaviors for ensuring file system consistency.

In order to handle the cascading dependencies, we record a blocking object to the descriptor of a process when the process is about to be blocked. There are two types of the blocking object in general: a process and an I/O for process dependency and I/O dependency, respectively. As a special case, a lock is recorded as the blocking object when a process should sleep to acquire the lock, in order to properly handle the cascading dependencies to both the lock owner and the waiters having higher lock-acquisition priority. Based on the recorded blocking object, a critical process can effectively track the cascading dependencies and can handle them using the process and I/O criticality inheritances.

Figure 3(c) demonstrates an example to describe how the cascading dependencies are handled: (1) critical pro-

cess P1 attempts to acquire the lock that is held by non-critical process P2. (2) Then, P2 inherits criticality from P1, and P1 checks P2's blocking object. (3) Since P2 is currently blocked waiting for an event from non-critical process P3, P3 also inherits criticality and P1 again checks P3's blocking object. (4) Due to P3 currently blocks on B1, P1 initiates reissuing of block B1 to NVWC and sleeps until the lock has been released by P2. (5) The I/O completion of the reissued B1 wakes up P3, and (6) P3 wakes up P2 after doing some residual work. (7) P2, in turn, wakes up P1 after completing its execution in the critical section. Finally, P1 enters the critical section and continues its job.

5 Implementation

We implemented our scheme on x86-64 Linux version 3.12. For the critical process identification, we added a pair of special priority values to dynamically set or clear criticality of a process (or a thread) to the existing `setpriority()` system call interface. We also added a field to the process descriptor for distinguishing criticality of each process.

For handling process and I/O dependencies, we implemented criticality inheritance protocols to blocking-based synchronization methods. Specifically, process and I/O criticality inheritances are implemented to the methods that synchronize with a process (e.g., `mutex_lock()`) and an ongoing I/O (e.g., `lock_buffer()`), respectively. In all the synchronization points, a blocking object is recorded into the descriptor of a process who is about to be blocked for synchronization.

We implemented our admission policy to FlashCache [2] version 3.1.1, which is a non-volatile block cache implemented as a kernel module. We modified the admission policy of FlashCache to cache only the writes synchronously requested by both critical and criticality-inherited processes. We also added the support for I/O criticality inheritance to FlashCache. In particular, the modified FlashCache maintains the list of outstanding non-critical writes to disk and searches the list when a critical process requests for reissuing a specific write. If the requested write is found in the list, FlashCache immediately reissues that write to NVWC and discards the result of the original write upon completion.

6 Application Studies

To validate the effectiveness of our scheme, we chose two widely deployed applications: PostgreSQL RDBMS [5] version 9.2 and Redis NoSQL store [23] version 2.8. For the critical process identification, we inserted eleven and two lines of code excluding comments to PostgreSQL and Redis, respectively. This result indicates that adopting the interface for critical process identification is trivial for typical data-intensive applications.

PostgreSQL RDBMS. In PostgreSQL, *backend* is dedicated to client for serving requests while other processes, such as *checkpointer*, *writer*, and *log writer*, carry out I/O jobs in background. The checkpointer flushes all dirty data buffers to disk and writes a special checkpoint record to the log file when the configured number of log files is consumed or the configured timeout happens, whichever comes first. The writer periodically writes some dirty buffers to disk to keep regular backend processes from having to write out dirty buffers. Similarly, the log writer periodically writes out the log buffer to disk in order to reduce the amount of synchronous writes needed for backend processes at commit time.

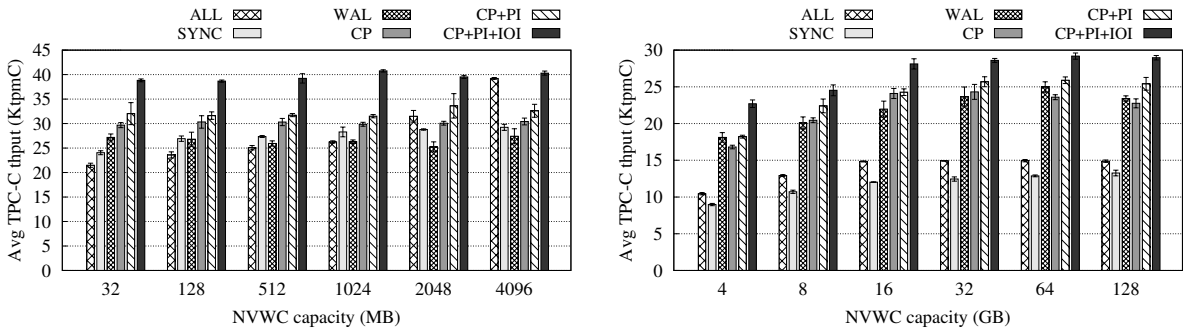
We classified backends as critical processes by calling the provided interface before starting the main loop of each backend. We also classified a process who is holding *WALWriteLock* as a temporary critical process because WALWriteLock is heavily shared between backends and other processes, and flushing the log buffer to a disk is conducted while holding the lock. This approach is similar to the priority ceiling [72] in that a process inherits criticality of a lock when the process acquires the lock.

Redis NoSQL store. Redis has two options to provide durability: snapshotting and command logging. The snapshotting periodically produces point-in-time snapshots of the dataset. The snapshotting, however, does not provide complete durability since up to a few minutes of data can be lost. The fully-durable command logging, on the other hand, guarantees the complete durability by synchronously writing an update log to a log file before responding back to the command. In the command logging, log rewriting is periodically conducted to constrain the size of the log file. Though the command logging can provide stronger durability than the snapshotting, it is still advisable to also turn the snapshotting on [25].

Similar to the PostgreSQL case, the snapshotting and log rewriting are conducted by child processes in background while a main server process serves all requests sequentially. Hence, we classified only the main server process as a critical process by calling the provided interface before starting the main event loop.

7 Evaluation

This section presents evaluation results based on the prototype implementation. We first detail the experimental environment. Then, we show the experimental results for both PostgreSQL and Redis to validate the effectiveness of the proposed scheme.



(a) Ramdisk NVWC (b) SSD NVWC

Figure 4: **PostgreSQL performance.** *TPC-C throughput is averaged over three runs for each admission policy.*

7.1 Experimental Setup

Our prototype was installed on Dell PowerEdge R420, equipped with two quad-core Intel Xeon E5-2407 2.4GHz processors and 16GB RAM; CPU clock frequency is set to the highest level for stable performance measurement. The storage subsystem is comprised of three 500GB 10K RPM WD VelociRaptor HDDs, one of which is dedicated to OS and the others are used as backing storage of NVWC. We used Ubuntu 14.04 with the modified Linux kernel version 3.12 as an OS and ext4 file system mounted with the default options.

For NVWC devices, we used a 4GB ramdisk (allocated from the main memory) and a 256GB Samsung 840 Pro SSD. To correctly emulate the persistency of the ramdisk-based NVWC in the existence of volatile CPU caches, we used the non-temporal memory copy described in [20] when the data is written to ramdisk. For the stable performance measurement of the SSD-based NVWC, we discard all blocks in SSD and give enough idle time before starting each experiment. In addition, in-storage volatile write cache was turned off to eliminate performance variations caused by internal buffering.

We used two criticality-oblivious admission policies: ALL and SYNC. ALL, which is the default of FlashCache, caches all incoming writes while SYNC caches only synchronous writes. In addition, we used three criticality-aware admission policies: CP, CP+PI, and CP+PI+IOI. CP caches synchronous writes requested by critical processes. CP+PI caches direct and cascading process dependencies-induced critical writes in addition to CP. CP+PI+IOI additionally caches direct and cascading I/O dependencies-induced critical writes.

7.2 PostgreSQL with TPC-C

We used TPC-C [7] as the realistic workload for PostgreSQL. We set TPC-C scale factor to ten, which corresponds to about 1GB of initial database, and simulated 24 clients running on a separate machine for 30 minutes. We report the number of New-Order transactions executed per minute (i.e., tpmC) as the performance metric. PostgreSQL was configured to have 512MB buffer pool, and the size of log files triggering checkpointing was set to 256MB. The database and log files are located on different HDDs according to the recommendation in the official document [6]. As the practical alternative of selective caching [21, 27, 42, 53], we used an additional policy denoted as WAL that caches all write traffics toward the log disk in NVWC. Since our work focuses on caching write I/Os, we eliminate read I/Os by warming up the OS buffer cache before starting the benchmark.

Performance with ramdisk-based NVWC. Figure 4(a) shows the TPC-C throughput averaged over three runs as the capacity of ramdisk-based NVWC increases from 32MB (scarce) to 4GB (sufficient). ALL achieves the lowest performance in the 32MB case because it stalls 58% of all writes. ALL gradually improves the performance as the NVWC capacity increases due to the reduction of write stalls. Note that the performance of ALL in the 4GB case is the optimal performance in our configuration because the capacity and bandwidth of NVWC are sufficient for absorbing all writes. SYNC slightly improves the performance compared to ALL in the low capacities since it reduces the number of write stalls by filtering out asynchronous writes. SYNC, however, cannot catch up the performance of ALL in the high capacities since it suffers from the dependencies induced by the asynchronous writes. Though WAL and CP do not suffer from write stalls at all in all the capacities, they achieve still lower performance than CP+PI and CP+PI+IOI due to runtime dependencies. CP+PI further improves performance by 4–12% over CP by handling process dependencies. CP+PI+IOI outperforms CP+PI by 18–29% by additionally handling I/O dependencies. Compared to ALL, CP+PI+IOI gains 80% performance improvement in the 32MB case and 72% reduction of cached writes without performance loss in the 4GB case.

To further analyze the reason behind the performance differences, we measured the wait time

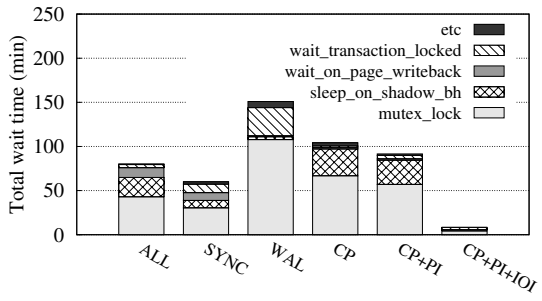


Figure 5: **Breakdown of PostgreSQL backends latency.** *512MB ramdisk is used as the NVWC device and network latencies are omitted for brevity.*

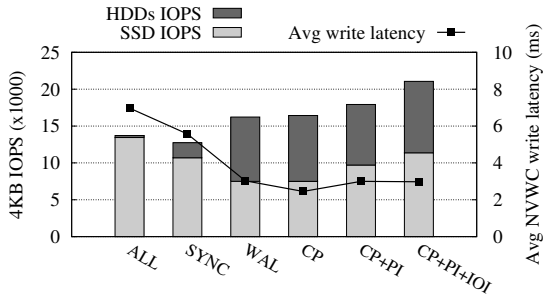


Figure 6: **Impact of queueing delay.** *16GB SSD is used as the NVWC device and synchronous write latency of critical processes is represented as NVWC write latency.*

of critical processes (i.e., PostgreSQL backends) in the 512MB NVWC case. As shown in Figure 5, ALL and SYNC incur the synchronous write latency (i.e., `wait_on_page_writeback`) and the mutex- and file system journaling-induced latencies (i.e., `sleep_on_shadow_bh` and `wait_transaction_locked`) described in Section 3, mainly due to frequent write stalls. Though WAL and CP mostly eliminate the synchronous write latencies by eliminating write stalls, they still incur excessive latencies mainly caused by the mutex and file system journaling. Though CP+PI further reduces latencies by resolving the mutex-induced dependency, it delays the progress of the critical processes because of unresolved I/O dependences. CP+PI+IOI eliminates most of the latencies since it additionally resolves I/O dependencies including the dependency to the journaling writes. As a result, CP+PI+IOI achieves the highest level of application performance in all the capacities.

Performance with SSD-based NVWC. Figure 4(b) shows the TPC-C throughput averaged over three runs as the capacity of SSD-based NVWC increases from 4GB to 128GB; the amount of concurrent I/Os for writeback decreases as the NVWC capacity increases. Unlike the case of the ramdisk-based NVWC, ALL achieves lower performance than the criticality-aware policies in all the

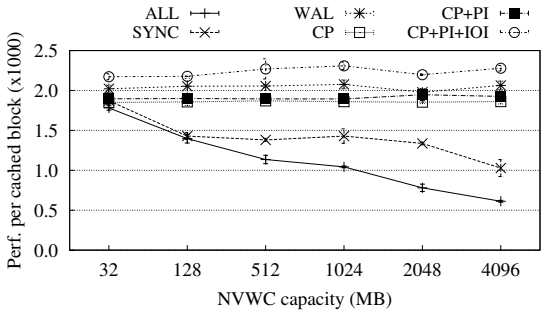


Figure 7: **Caching efficiency.** *Ramdisk is used as the NVWC device.*

capacities due to severe congestion in the request queues of the SSD. Though SYNC can ease the contention in the SSD more than ALL, it still has dependency problems incurred by the asynchronous writes directly routed to HDDs. On the other hand, WAL and the criticality-aware policies improve the application performance by reducing queue congestion compared to both ALL and SYNC. In particular, CP+PI+IOI outperforms ALL and SYNC by about 1.8–2.2× and 2.2–2.5×, respectively, because it minimizes the queueing delays of critical writes by filtering out more than a half of writes while effectively handling process and I/O dependencies.

To show the impact of queueing delay on critical writes, we measured 4KB IOPS for both the SSD and HDDs, and the average latency of synchronous writes requested by backends. As shown in Figure 6, ALL and SYNC utilize the SSD better than the other policies. This high utilization, however, causes severe congestion in the request queues of SSD, thereby delaying the processing of critical writes. WAL and the criticality-aware policies utilize both the SSD and the HDDs in a more balanced manner, thereby decreasing the queueing delay of critical writes.

Caching efficiency. In order to quantify the caching efficiency in terms of application performance, Figure 7 plots the performance per cached block as the capacity of the ramdisk-based NVWC increases. WAL and the criticality-aware policies show higher caching efficiencies compared to ALL and SYNC. Note that ALL and SYNC unexpectedly show high caching efficiency in the low NVWC capacities because FlashCache directs a write I/O to backing storage instead of waiting for a free block when there is no free block in NVWC. Overall, CP+PI+IOI utilizes NVWC more efficiently by 1.2–3.7× and 1.2–2.2× compared to ALL and SYNC, respectively.

Breakdown of critical writes. To help understand which types of data and I/O constitute critical writes, Table 3 shows the breakdown of critical writes in terms of data and I/O types. As we expect, the dominant type of data comprising critical writes is the logs that are synchronously written by backends during transaction com-

	Ratio (%)	CP	PI	IOI	Total
Data	Data (DB)	0	0.5209	0.0515	0.5723
	Data (LOG)	98.7409	0.5707	0.0007	99.3123
	Metadata	0	0	0.0014	0.0014
	Journal	0	0.0782	0.0357	0.1140
	Total	98.7409	1.1698	0.0893	100
I/O	Sync.	98.7409	1.0312	0.0357	99.8078
	Async.	0	0.1387	0.0535	0.1922
	Total	98.7409	1.1698	0.0893	100

Table 3: **Breakdown of critical writes.** *4GB ramdisk is used as the NVWC device in the case of CP+PI+IOI.*

mits. However, the rest of the critical writes is still crucial since it contributes to additional 32% performance improvement over CP alone (Figure 4(a)). On the other side, the dominant type of I/O comprising critical writes is synchronous writes. Though the portion of asynchronous writes is insignificant, it contributes to additional 38% performance improvement over SYNC (Figure 4(a)). Overall, dependency-induced critical writes have significant impact on application performance.

Performance disparity. Interestingly, we found the disparity between the system performance (i.e., IOPS) and the application performance (i.e., tpmC). For instance, as shown in Figure 8, ALL better utilizes NVWC by 40% than CP+PI+IOI leading to achieve 10% higher system performance in the 512MB ramdisk case. However, CP+PI+IOI accomplishes 57% higher application performance than that of ALL because CP+PI+IOI avoids write stalls in the critical paths. This result validates our argument on the necessity of the request-oriented approach in order to effectively utilize a given NVWC device.

7.3 Redis with YCSB

For Redis, we used the update-heavy (Workload A) and read-mostly (Workload B) workloads provided by the YCSB benchmark suite [26]. The data set was composed of 0.5 million objects each of which is 1KB in size. We simulated 40 clients running on a separate machine to generate ten millions of operations in total. We report operations per second (i.e., ops/sec) as the performance metric. We enabled both snapshotting and command logging according to the suggestion in the official document [25]. Due to the single threaded design of Redis [24], we concurrently ran four YCSB benchmarks against four Redis instances to utilize our multi-core testbed.

Performance. Figure 9 demonstrates the average YCSB throughput over three runs normalized to ALL. SYNC improves the performance over ALL since it filters out the asynchronous writes issued by the kernel thread that cleans the OS buffer cache. Unlike the case of PostgreSQL, CP shows significantly low performance

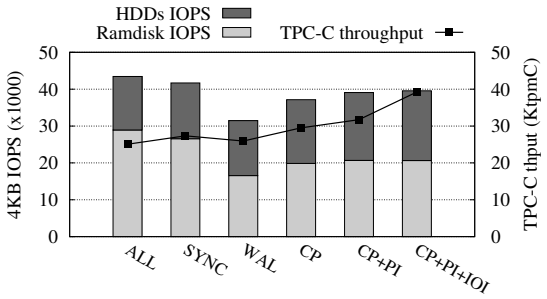


Figure 8: **Performance disparity.** *512MB ramdisk is used as the NVWC device.*

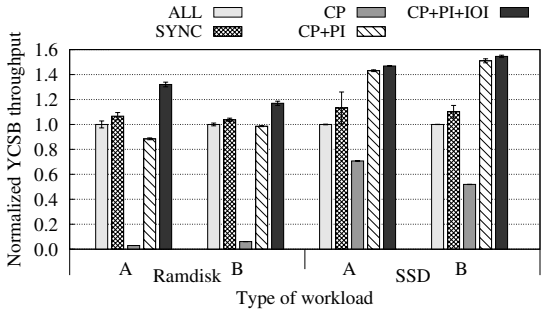


Figure 9: **Redis performance.** *512MB ramdisk and 16GB SSD are used as the NVWC devices.*

compared to the other policies because Redis frequently incurs synchronous writes conducted by a journaling daemon, which cannot be detected as the critical process by CP, within the critical path of update request. CP+PI dramatically improves the performance by 2–30× over CP because it additionally caches the journaling writes when there is a dependency between a critical process and the journaling daemon. CP+PI+IOI further improves the performance by 3–49% over CP+PI by additionally resolving the I/O dependencies mainly incurred by the synchronizations to guarantee the file system consistency. Though Workload B mostly consists of read requests, the performance is affected by the admission policy used. This is because Redis serves the requests from all clients sequentially using a single thread, thereby delaying the processing of read requests that are queued behind update requests. By providing the first class support for critical writes, CP+PI+IOI outperforms ALL by 17–32% and 47–55% while reducing cached write by 20–29% and 84–87% in the ramdisk- and SSD-based NVWC, respectively.

Tail latency. To show the impact of the admission policies on tail latency, we present the latency distribution of YCSB requests in the SSD-based NVWC case. As shown in Table 4, only CP+PI+IOI keeps 99.9^{th} and 99.99^{th}-percentile latencies below 100ms, which makes users feel more responsive than higher latencies [18]. ALL and SYNC, on the other hand, increase the 99.9^{th}-

	Latency (ms)	99th-%ile	99.9th-%ile	99.99th-%ile
A	ALL	80	649	>1000
	SYNC	72	678	>1000
	CP+PI+IOI	32	50	79
B	ALL	70	572	>1000
	SYNC	59	438	>1000
	CP+PI+IOI	23	32	83

Table 4: **Redis tail latency.** *16GB SSD is used as the NVWC device.*

percentile latency by an order of magnitude compared to that of CP+PI+IOI. Moreover, the 99.99^{th}-percentile latencies of ALL and SYNC exceeds one second, which is the maximum latency reported by YCSB. Considering the significance of providing consistent response latencies to users [16, 71] especially for large-scale services [29, 75], this result indicates that the proposed scheme is essential for providing high quality services to users.

8 Related Work

Non-volatile cache. A large volume of work has been done to efficiently utilize non-volatile caches based on NV-DRAM [15, 36, 37, 41], flash memory [19, 62, 70], and SCM [14, 33, 48, 56, 61]. In addition, the case of client-side non-volatile caches has been widely explored for networked storage systems [10, 11, 49, 65]. On the other side, researchers have extensively investigated the case of non-volatile cache optimized for database systems such as cost- and pattern-aware flash caches for relational database [32, 46, 55, 58] and persistent key-value store [30]. In addition, several studies have been investigated the case of dedicating NV-DRAM [27, 42] and flash memory [21, 53] to buffer or store transaction logs of relational databases. None of the previous work has taken the context of request execution into account for managing a non-volatile cache despite of its importance.

I/O classification. Prioritizing synchronous I/Os over asynchronous ones has been known as a reasonable method for improving system performance [28, 35]. Classifying I/Os based on explicit hints from data-intensive applications has been well-studied. Li *et al.* [54] proposed a cache replacement policy that exploits different write semantics in a relational database to maintain exclusivity between storage server and client caches. Later, Xin *et al.* [79] proposed a more general framework for the client hint-based multi-level cache management. Similarly, Mesnier *et al.* [60] proposed an I/O classification interface between computer and storage systems. For user-interactive desktop environments, Redline [76] statically gives higher I/O priority to interactive applications over non-interactive ones. Unlike the previous work, our classification scheme considers the I/O priority as dynamic property since it can be changed during runtime due to complex dependencies.

Request tracing. Request-oriented performance debugging has been widely explored for the end-user experience. Instrumentation-based profilers such as Project 5 [9] and MagPie [12] have been used for tracking request flows triggered by user requests. The Mystery Machine [22] and the lprof tool [82] extract the per-request performance behaviors from the log files to diagnose performance problems in large-scale distributed systems. In addition, Shen has studied architectural implications of request behavior variations in modern computer systems [73]. For user-interactive mobile platforms, AppInsight [68] and Panappticon [81] provide the information on the critical path of user request processing to application developers for improving user-perceived responsiveness. In this work, we focus on tracking request execution in the write I/O path and apply the acquired information to the admission policy of NVWC for improving application performance.

9 Conclusion and Future Direction

We present the request-oriented admission policy, which selectively caches the writes that eventually affect the application performance while preventing unproductive writes from occupying and wearing-out capacity-constrained NVWCs. The proposed scheme can contribute to reducing capital cost of expensive NVWCs satisfying desired service-level objectives. The results from the in-depth analysis on realistic workloads justify our claim that storage systems should consider the context of request execution to guarantee a high degree of application performance.

We plan to develop automatic critical process identification at kernel-level without an application hint in order to support legacy and proprietary applications. We also plan to apply the proposed classification to interactive systems, such as mobile systems, considering a direct user input as an external request.

10 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Phillipa Gill, for their valuable comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2014R1A2A1A10049626).

References

[1] Everspin Spin-Torque MRAM. `http://www.everspin.com/spinTorqueMRAM.php`.

[2] FlashCache. `https://github.com/facebook/flashcache`.

[3] Micron annouces availability of phase change memory for mobile devices. http://investors.micron.com/releasedetail.cfm?ReleaseID=692563.

[4] MySQL 5.7 reference manual. http://dev.mysql.com/doc/refman/5.7/en/threads-table.html.

[5] PostgreSQL. http://www.postgresql.org.

[6] PostgreSQL WAL internals. http://www.postgresql.org/docs/9.2/static/wal-internals.html.

[7] The TPC-C benchmark. http://www.tpc.org/tpcc.

[8] AGIGATECH. AGIGRAM (TM) Non-Volatile System. http://www.agigatech.com/agigaram.php.

[9] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles - SOSP '03*.

[10] ARTEAGA, D., AND ZHAO, M. Client-side flash caching for cloud systems. In *Proceedings of the International Conference on Systems and Storage - SYSTOR '14*.

[11] BAKER, M., ASAMI, S., DEPRIT, E., OUSETERHOUT, J., AND SELTZER, M. Non-volatile memory for fast, reliable file systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '92*.

[12] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation - OSDI '04*.

[13] BHADKAMKAR, M., GUERRA, J., USECHE, L., BURNETT, S., LIPTAK, J., RANGASWAMI, R., AND HRISTIDIS, V. BORG: Block-reORGanization for self-optimizing storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies - FAST '09*.

[14] BHASKARAN, M. S., XU, J., AND SWANSON, S. Bankshot: Caching slow storage in fast non-volatile memory. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads - INFLOW '13*.

[15] BISWAS, P., RAMAKRISHNAN, K. K., AND TOWSLEY, D. Trace driven analysis of write caching policies for disks. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '93*.

[16] BRUTLAG, JAKE. Speed matters for Google Web search. http://googleresearch.blogspot.kr/2009/06/speed-matters.html.

[17] BURR, G. W., KURDI, B. N., SCOTT, J. C., LAM, C. H., GOPALAKRISHNAN, K., AND SHENOY, R. S. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development 52*, 4.5 (July 2008), 449–464.

[18] CARD, S. K., ROBERTSON, G. G., AND MACKINLAY, J. D. The information visualizer: An information workspace. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems - CHI '91*.

[19] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing - ICS '11*.

[20] CHEN, F., MESNIER, M. P., AND HAHN, S. A protected block device for persistent memory. In *Proceedings of the 30th Symposium on Mass Storage Systems and Technologies - MSST '14*.

[21] CHEN, S. FlashLogging: Exploiting flash devices for synchronous logging performance. In *Proceedings of the 35th SIGMOD International Conference on Management of Data - SIGMOD '09*.

[22] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. The Mystery Machine: End-to-end performance analysis of large-scale Internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation - OSDI '14*.

[23] CITRUSBYTE. Redis. http://redis.io/.

[24] CITRUSBYTE. Redis latency problems troubleshooting. http://redis.io/topics/latency.

[25] CITRUSBYTE. Redis persistence. http://redis.io/topics/persistence.

[26] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing - SoCC '10*.

[27] COPELAND, G., KELLER, T., KRISHNAMURTHY, R., AND SMITH, M. The case for safe RAM. In *Proceedings of the 15th International Conference on Very Large Data Bases - VLDB '89*.

[28] CORBET, J. Solving the ext3 latency problem. http://lwn.net/Articles/328363/.

[29] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (Feb. 2013), 74.

[30] DEBNATH, B., SENGUPTA, S., AND LI, J. FlashStore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment 3*, 1-2 (Sept. 2010), 1414–1425.

[31] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles - SOSP '07*.

[32] DO, J., ZHANG, D., PATEL, J. M., DEWITT, D. J., NAUGHTON, J. F., AND HALVERSON, A. Turbocharging DBMS buffer pool using SSDs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data - SIGMOD '11*.

[33] DOH, I. H., LEE, H. J., MOON, Y. J., KIM, E., CHOI, J., LEE, D., AND NOH, S. H. Impact of NVRAM write cache for file system metadata on I/O performance in embedded systems. In *Proceedings of the 2009 ACM Symposium on Applied Computing - SAC '09*.

[34] EDGE, JAKE. Finding system latency with LatencyTOP. http://lwn.net/Articles/266153/.

[35] GANGER, G. R., AND PATT, Y. N. The process-flow model: Examining I/O performance from the system's point of view. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '93*.

[36] GILL, B. S., KO, M., DEBNATH, B., AND BELLUOMINI, W. STOW: A spatially and temporally optimized write caching algorithm. In *Proceedings of the 2009 USENIX Annual Techinal Conference - USENIX '09*.

[37] GILL, B. S., AND MODHA, D. S. WOW: Wise ordering for writes - combining spatial and temporal locality in non-volatile caches. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies - FAST '05*.

[38] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1992.

[39] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro '09*.

[40] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies - FAST '12*.

[41] HAINING, T. R., AND LONG, D. D. E. Management policies for non-volatile write caches. In *Proceedings of the 1999 IEEE International Performance, Computing and Communications Conference - IPCCC '99*.

[42] HEISER, G., LE SUEUR, E., DANIS, A., BUDZYNOWSKI, A., SALOMIE, T.-L., AND ALONSO, G. RapiLog: Reducing system complexity through verification. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*.

[43] HEWITT, E. *Cassandra: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[44] HUANG, P., WU, G., HE, X., AND XIAO, W. An aggressive worn-out flash block management scheme to alleviate SSD performance degradation. In *Proceedings of the 9th European Conference on Computer Systems - EuroSys '14*.

[45] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems - HotStorage '14*.

[46] KANG, W.-H., LEE, S.-W., AND MOON, B. Flash-based extended cache for higher throughput and faster recovery. *Proceedings of the VLDB Endowment 5*, 11 (July 2012), 1615–1626.

[47] KGIL, T., ROBERTS, D., AND MUDGE, T. Improving NAND flash based disk caches. In *Proceedings of the 35th Annual International Symposium on Computer Architecture - ISCA '08*.

[48] KIM, H., SESHADRI, S., DICKEY, C. L., AND CHIU, L. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies - FAST '14*.

[49] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies - FAST '13*.

[50] KOLLER, RICARDO AND RANGASWAMI, R. I/O deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies - FAST '10*.

[51] LAMPSON, B. W., AND REDELL, D. D. Experience with processes and monitors in Mesa. *Communications of the ACM 23*, 2 (Feb. 1980), 105–117.

[52] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th annual international symposium on Computer architecture - ISCA '09*.

[53] LEE, S.-W., MOON, B., PARK, C., KIM, J.-M., AND KIM, S.-W. A case for flash memory SSD in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data - SIGMOD '08*.

[54] LI, X., ABOULNAGA, A., SALEM, K., SACHEDINA, A., AND GAO, S. Second-tier cache management using write hints. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies - FAST '05*.

[55] LIU, X., AND SALEM, K. Hybrid storage management for database systems. *Proceedings of the VLDB Endowment 6*, 8 (June 2013), 541–552.

[56] LIU, Z., WANG, B., CARPENTER, P., LI, D., VETTER, J. S., AND YU, W. PCM-based durable write cache for fast disk I/O. In *Proceedings of the IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems - MASCOTS '12*.

[57] LOWELL, D. E., AND CHEN, P. M. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles - SOSP '97*.

[58] LUO, T., LEE, R., MESNIER, M., CHEN, F., AND ZHANG, X. hStorage-DB: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proceedings of the VLDB Endowment 5*, 10 (June 2012), 1075–1087.

[59] MALVIYA, N., WEISBERG, A., MADDEN, S., AND STONEBRAKER, M. Rethinking main memory OLTP recovery. In *Proceedings of the 30th IEEE International Conference on Data Engineering - ICDE '14*.

[60] MESNIER, M., CHEN, F., LUO, T., AND AKERS, J. B. Differentiated storage services. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles - SOSP '11*.

[61] MILLER, E. L., BRANDT, S. A., AND LONG, D. D. HeRMES: High-performance reliable MRAM-enabled storage. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems - HOTOS '01*.

[62] OH, Y., CHOI, J., LEE, D., AND NOH, S. H. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies - FAST '12*.

[63] OUSTERHOUT, J., AND DOUGLIS, F. Beating the I/O bottleneck: A case for log-structured file systems. *ACM SIGOPS Operating Systems Review 23*, 1 (Jan. 1989), 11–28.

[64] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles - SOSP '95*.

[65] QIN, D., BROWN, A. D., AND GOEL, A. Reliable writeback for client-side flash caches. In *Proceedings of the 2014 USENIX Annual Technical Conference - USENIX '14*.

[66] QURESHI, M. K., KARIDIS, J., FRANCESCHINI, M., SRINIVASAN, V., LASTRAS, L., AND ABALI, B. 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 - Micro '09*.

[67] RAOUX, S., BURR, G. W., BREITWISCH, M. J., RETTNER, C. T., CHEN, Y.-C., SHELBY, R. M., SALINGA, M., KREBS, D., CHEN, S.-H., LUNG, H.-L., AND LAM, C. H. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development 52*, 4.5 (July 2008), 465–479.

[68] RAVINDRANATH, L., PADHYE, J., AGARWAL, S., MAHAJAN, R., OBERMILLER, I., AND SHAYANDEH, S. AppInsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation - OSDI '12*.

[69] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference - USENIX '00*.

[70] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. FlashTier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*.

[71] SCHURMAN, E., AND BRUTLAG, J. *The user and business impact of server delays, additional bytes, and HTTP chunking in Web search.* O'Reilly Velocity, 2009.

[72] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers 39*, 9 (1990), 1175–1185.

[73] SHEN, K. Request behavior variations. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '10*.

[74] SOLWORTH, J. A., AND ORJI, C. U. Write-only disk caches. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data - SIGMOD '90*.

[75] TIMOTHY, Z., TUMANOV, A., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. PriorityMeister: Tail latency QoS for shared networked storage. In *Proceedings of the 2014 ACM Symposium on Cloud Computing - SoCC '14*.

[76] TING YANG, TONGPING LIU, EMERY D. BERGER, SCOTT F. KAPLAN, J. ELIOT, B. M. Redline: First class support for interactivity in commodity operating systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation - OSDI '08*.

[77] VERMA, A., KOLLER, R., USECHE, L., AND RANGASWAMI, R. SRCMap: Energy proportional storage using dynamic consolidation. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies - FAST '10*.

[78] VIKING TECHNOLOGY. ArxCis-NV (TM) Non-Volatile Memory Technology. `http://www.vikingtechnology.com/arxcis-nv`.

[79] XIN LIU, ASHRAF ABOULNAGA, XUHUI LI, K. S. CLIC: Client-informed caching for storage servers. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies - FAST '09*.

[80] YANG, J., PLASSON, N., GILLIS, G., AND TALAGALA, N. HEC: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Conference on Systems and Storage - SYSTOR '13*.

[81] ZHANG, L., BILD, D. R., DICK, R. P., MAO, Z. M., AND DINDA, P. Panappticon: Event-based tracing to measure mobile application and platform performance. In *Proceedings of the 2013 International Conference on Hardware/Software Codesign and System Synthesis - CODES+ISSS '13*.

[82] ZHAO, X., ZHANG, Y., LION, D., FAIZAN, M., LUO, Y., YUAN, D., AND STUMM, M. lprof: A nonintrusive request flow profiler for distributed systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation - OSDI '14*.

[83] ZHENG, H., AND NIEH, J. RSIO: Automatic user interaction detection and scheduling. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '10*.

[84] ZHENG, H., AND NIEH, J. SWAP: A scheduler with automatic process dependency detection. In *In Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation - NSDI '04*.