Design, Implementation, and Evaluation of Write-Back Policy with Cache Augmented Data Stores

Shahram Ghandeharizadeh and Hieu Nguyen
Database Laboratory Technical Report 2018-07
Computer Science Department, USC
Los Angeles, California 90089-0781
{shahram,hieun}@usc.edu

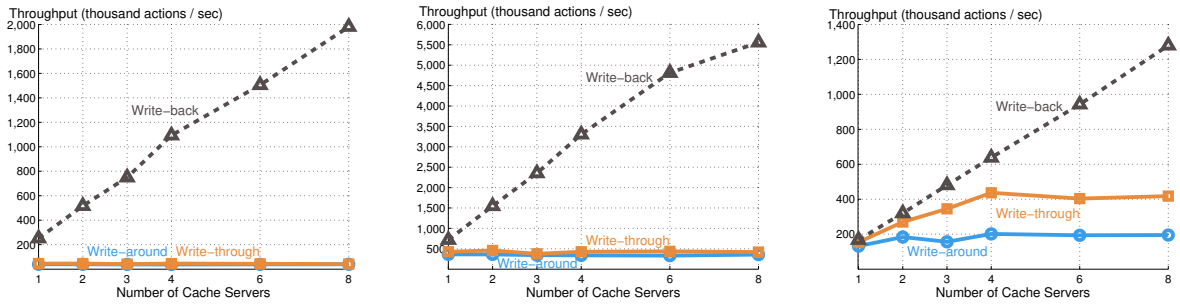
Abstract

The Cache Augmented Data Store (CADS) architecture extends a persistent data store with an in-memory cache manager. It is widely deployed to support read-intensive workloads. However, its write-around and write-through policies prevent the caching tier from absorbing write load. This means the data store layer must scale to process writes even when the extra capacity is not needed for read load. We address this limitation by devising a write-back technique to enable the caching layer to process both reads and writes. This technique preserves ACID transactions. We present a client side implementation of write-back and evaluate it using the YCSB, BG, and TPC-C benchmarks. Obtained results show the write-back policy enables the caching layer to process writes and scale horizontally. With some workloads, it outperforms write-around and write-through by more than 50x. In addition, we compare our write-back with (a) write-back policy of a data store such as MongoDB and (b) write-back policy of a host-side cache such as Flashcache. Our proposed write-back complements these alternatives, enhancing their performance more than 2x.

1 Introduction

Cache Augmented Data Stores (CADSs) have been widely adopted for workloads that exhibit a high read to write ratio. Examples include social networking sites such as Facebook, Twitter and LinkedIn [18, 19, 1, 32]. CADSs extend a persistent data store with a caching layer using off-the-shelf commodity servers. Redis and memcached are popular in-memory key-value stores used as cache managers.

Write-around (invalidation) and write-through (refill) policies apply a write to the data store synchronously. This prevents the caching tier from absorbing writes, requiring the data store layer to scale to process writes even when its extra capacity is not required for read load. This study presents a client-side implementation of write-back (write-behind) policy to address this limitation. The proposed technique buffers writes in the cache and applies them to the data store asynchronously.



(a) Workload A (b) Workload B (c) Workload S

Figure 1: Throughput of different YCSB workloads with write-around, write-through and write-back policies.

Write-back enhances both the performance and horizontal scalability of CADSs significantly. To illustrate, Figure 1 shows the scalability of a CADS configuration consisting of one MongoDB server as we vary the number of servers in its caching layer from one to eight. We show results for several Yahoo! Cloud Serving Benchmark [9] (YCSB) workloads: the write heavy Workload A and read heavy Workloads B and S. With write-around and write-through, the caching layer does not scale because the data store is the bottleneck. With write-back, the throughput scales almost linearly even though the data store remains fully utilized. This is because write-back buffers writes in the caching layer and applies them to the data store asynchronously, removing the data store from the critical path of processing requests.

These results explain why caching middleware such as Oracle Coherence [33], EhCache [40] and Infinispan [23] support the write-back (*write-behind*) policy. They do so by providing simple interfaces of a key-value store such as get, put and delete. A developer is responsible for providing an application specific implementation of these interfaces.

Design and implementation of a write-back policy must address several challenges. First, how to represent data store writes (termed *buffered writes*) as cache entries and how to prevent them from being evicted by the cache replacement policy. Second, how to apply the buffered writes from the cache to the data store efficiently and ensure read-after-write consistency. Reads must always observe values produced by the latest writes even when they are buffered in the cache. Otherwise, they may produce stale results that impact the correctness of application and pollute the cache. Third, how to provide durability of writes in the presence of cache failures. If a write is acknowledged and its buffered writes are lost due to a cache server failure then the write is no longer durable. Fourth, how to process a non-idempotent buffered write such as increment without compromising consistency in the presence of failures.

This study presents a write-back technique that addresses the above challenges. Our proposed technique provides:

- **Read-after-write consistency.** A read is guaranteed to observe the latest writes. A read that references a cache entry with pending buffered writes is processed in the context of these writes. We also use leases to prevent undesirable race conditions.

- **High performance and scalability.** Our implementation of write-back scales the system throughput as we increase the number of cache servers. This is achieved by partitioning buffered writes across cache servers and using the client component of the caches to apply these writes to the data store asynchronously.

- **Durability.** Buffered writes are pinned in memory, preventing their eviction by the cache replacement policy. Moreover, we replicate buffered writes across 3 or more cache servers to tolerate cache server failures. These replicas may be assigned to servers in different racks within a data center. To tolerate data center failure, one may use non-volatile memory such as today's NVDIMM-N [36].

We assume correctness of a write is the responsibility of the application developer. It transitions the database from one consistent state to another [20]. (Transaction processing systems [20] make the same assumption.) Key elements of our design include:

1. To use a write-back policy, an application must provide a *mapping* of cache entries to buffered writes. This enables a cache miss by a read to identify pending writes (if any) that impact the missing value. Using these pending writes, the application may compute the missing value and provide read-after-write consistency.

2. Use leases to provide read-after-write consistency and enhance data availability in the presence of failures.

3. Store buffered writes and their mappings in the caching layer by pinning them. With atomic operations (termed sessions) that impact more than one key-value pair, we capture their spatial relationship in the buffered writes. Similarly, we capture the temporal relationship between sessions that impact the same key-value pairs to apply buffered writes to the data store in the same order.

4. Differentiate between idempotent and non-idempotent changes. We present two different techniques to support non-idempotent writes with NoSQL and SQL systems.

5. Replicate buffered writes across multiple cache servers to enhance availability in the presence of failures.

6. Partition buffered writes to balance load across multiple cache servers and use client component of the cache to minimize server software complexity (load) and prevent single source of bottleneck.

While all caching middleware advocate partitioning of buffered writes for scalability and their replication for high availability as best practices [33, 40, 23, 22], they lack the first four aforementioned design elements: mappings, leases, spatial and temporal relationships of buffered writes, and support for non-idempotent changes. These differences make it challenging to compare our technique with the existing caching middleware, see Section 5.

This paper makes several *contributions*. First, we introduce the concept of mapping and use of leases to provide read-after-write consistency with cache misses. Second, we support non-idempotent changes with both No-SQL and SQL systems. Third, we present an implementation of these concepts in two different designs. While Design 1 is appropriate for a document (relational) store that provides ACID properties at the granularity of one document

(row), Design 2 supports complex transactions consisting of multiple statements impacting multiple documents (rows) or cache entries. Fourth, we provide a comprehensive evaluation of a client-side implementation of write-back using off-the-shelf software components with YCSB [9], BG [5], and Transaction Processing Performance Council (TPC) Benchmark C [42] (TPC-C) benchmarks. Obtained results show write-back enhances performance with all benchmarks as long as memory is abundant, out-performing both write-through and write-around polices.

We also evaluate the impact of the following factors on write-back performance: the number of background threads (BGTs), maximum amount of pinned memory for buffered writes, and degree of replication for buffered writes. Increasing the number of BGTs reduces the amount of memory required by write-back. However, it also reduces the throughput observed by the application.

We compare the write-back policy with those used in host-side caches and caches of data stores such as MongoDB, i.e., MongoDB configured with writeConcern set to ACKNOWLEDGED. The proposed write-back policy complements its block-based alternatives, host-side caches [7, 13, 21, 25] and MongoDB's write-back technique, enhancing their performance several folds.

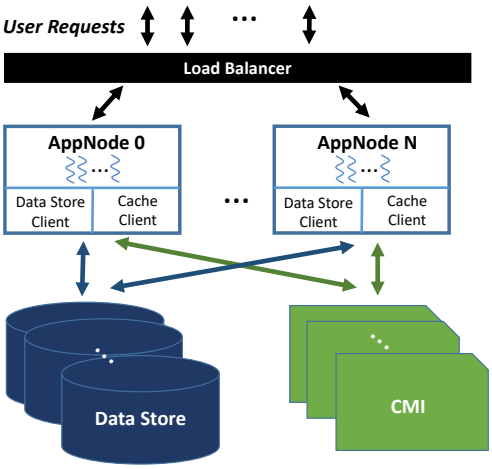


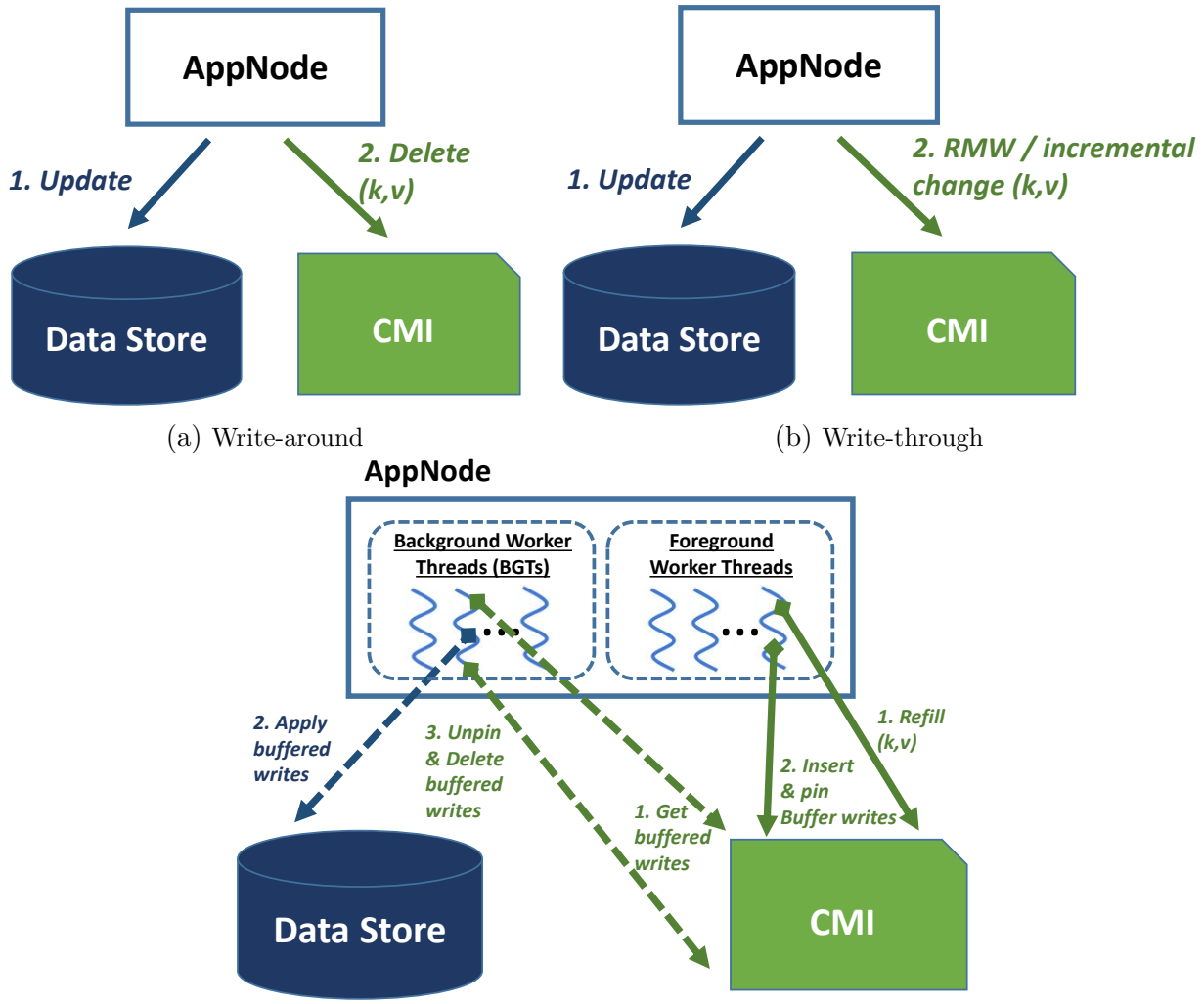
Figure 2: Architecture of a Cache Augmented Data Store.

Write-back has two limitations. First, it is more complex to implement than either write-through or write-around policies, requiring additional software. Second, its performance with a limited amount of memory may be inferior when compared with the other policies. Specifically, we observe calcification [24] of memory with memcached when the size of mappings and buffered writes is different. This prevents write-back from utilizing its maximum allocated pinned memory.

The rest of this paper is organized as follows. Section 2 provides an overview of the CADS architecture. Sections 3 presents two designs for write-back. Section 4 evaluates the write-back policy by comparing it with other write policies and block-based caches. Section 5 describes related work. Section 6 provides brief conclusions and outlines our future research directions.

2 Architecture Overview

Figure 2 shows the architecture of a CADS. A load balancer directs user requests to different application servers. Each application server consists of one or many AppNode instances serving user requests. Each AppNode has client components to communicate with the persistent data store (e.g., JDBC with SQL systems) and the cache (e.g., Whalin Client with memcached). The data store may either be a SQL (e.g., MySQL [31], PostgreSQL [41], OracleDB [34]) or a NoSQL data store (e.g., MongoDB [30], CouchBase [10]). Multiple cache manager instances (CMIs) may be deployed on a cache server. Example CMIs includes an



(a) Write-around (b) Write-through

(c) Write-back

Figure 3: Write-around, write-through and write-back policies. Solid arrows are performed in the foreground. Dashed arrows are performed in the background.

in-memory key-value store such as *memcached* [2] or *Redis* [35]. AppNodes communicate with CMIs via message passing. These cache managers provide simple interfaces such as `get`, `set`, `append` and `delete`.

Definition 2.1. *A cache entry is represented as a key-value pair* (k_i, v_i) *where* k_i *identifies the entry and* v_i *is the value of the entry. Both* k_i *and* v_i *are application specific and authored by a developer.*

An AppNode read identifies the key k_i and gets its value v_i from the cache. If the value v_i exists then the read has observed a cache hit and proceeds to consume v_i. Otherwise, with a cache miss, the read queries the data store for the data item. It may fetch a few rows of a relational database or a document of a document store to compute the key-value pair to be inserted in the cache for future reference. Reads that observe a hit for these key-value pairs benefit because result look-up from the in-memory cache is much faster than query

processing using the data store [18, 32].

Figure 3 shows the write-around (invalidate) and write-through (refill) policies. Both update the data store synchronously. While write-around deletes the impacted key-value pairs, write-through updates them. Write-through may employ incremental update (e.g., append or increment) or read-modify-write to update a key-value pair. With write-back, see Figure 3c, a write updates the impacted key-value pairs similar to write-through. However, it stores one or more replicas of its changes (termed *buffered write*) in the CMI instead of applying it to the data store. The write is then acknowledged to the user as successful. Background threads, BGTs, apply the buffered write to the data store asynchronously. Figure 3c shows these threads are co-located with AppNode. However, this is not a requirement and a different process may host these threads.

We make several *assumptions* about the cache manager that may not be standard. First, the application may *pin* and *un-pin* a key-value pair when setting it in a CMI. This means the CMI's cache replacement technique may not evict a pinned key-value pair. We pin buffered writes and their mappings in one or more CMIs. A background thread that applies a buffered write to the data store un-pins and deletes them.

Second, we assume the concept of sessions. A session is an atomic operation with a unique identifier. It reads and writes one or more cache entries and issues one transaction to the data store. We use a session to implement a transaction of the TPC-C benchmark. A session that reads a key-value pair must obtain a Shared (S) lease on it prior to reading it. A session that writes a key-value pair must obtain an eXclusive (X) lease on it prior to updating[1] its value.

Requested Lease	Existing Lease	
	S	X
S	Grant S lease	Abort and Retry
X	Grant X and void S lease	Abort and Retry

Table 1: S and X Lease Compatibility.

The S and X leases are different than read and write locks in several ways. First, S and X leases are non-blocking. As shown in the compatibility Table 1, when a session T_r requests a S lease on a key-value pair with an existing X lease, it aborts and retries. Second, when a session T_r requests an X lease on a data item with an existing S lease granted to session T_h, T_r wounds T_h by voiding its S lease. At its subsequent request, T_h is notified to abort and restart. This prevents write sessions from starving. Third, they have a finite lifetime in the order of hundreds of milliseconds. Once they expire, their referenced key-value pair is deleted. This is suitable for a distributed environment where an application node fails causing its sessions holding leases to be lost. Leases of these sessions expire after sometime to make their referenced data items available again.

Prior to committing its data store transaction, a session validates itself to ensure all its leases are valid. Once a session is validated, its S leases become golden. This means they may no longer be voided by an X lease. An X lease that encounters a golden S lease [3] is

[1] An update may be in the form of a read-modify-write or incremental update such as increment, append, etc.

forced to abort and retry. Once the session commits its database transaction, it commits the session and releases its leases.

Different components of a distributed CADS may implement the write-back policy and its buffered writes. For example, it may be implemented by the application instances, CMIs, or a middleware between the application instances and CMIs, or a hybrid of these. This paper describes an implementation using the application instances, see Figure 3c.

3 Write-back: 2 Designs

This section presents two different designs for the write-back policy. They assume the data store transaction that constitutes a session has different complexity. Design 1 assumes simple data store transactions that either read or write a single document (row) of a document (relational) store such as MongoDB (MySQL). It is suitable for workloads modeled by the YCSB benchmark. Design 2 assumes a session's data store transaction reads and/or writes multiple rows of a SQL (or documents of a transactional MongoDB) data store. It is suitable for complex workloads such as those modeled by the TPC-C benchmark.

Design 1 maintains changes at the granularity of each document. A cache look up that observes a miss is provided with a *mapping* that identifies changes that should be used when computing the missing cache entry. To enable BGTs to discover and apply changes to the data store asynchronously, it maintains a list of documents with changes, termed PendingWrites.

Design 2 maintains changes at the granularity of a session. Its mapping identifies the dependence of a cache entry on the sessions with pending changes. A cache miss uses this mapping to identify sessions with pending buffered writes that should be considered when computing the missing cache entry. It maintains a queue that specifies the order in which sessions commit. A BGT uses this queue to discover sessions with pending writes and applies them to the data store in the same order as their serial commit.

Design 2 is different than Design 1 in that its queue identifies temporal dependence of the sessions across multiple data items in the data store. Design 1 is simpler because it maintains the order of changes per document by assuming a session writes only one document. Design 2 is essential for preserving SQL's integrity constraints such as foreign key dependencies between rows of different tables.

Below, we provide a formal definition of a change, a buffered write, and a mapping. Subsequently, we describe PendingWrites and queues to facilitate discovery of buffered writes by BGTs. Finally, we present BGTs and how they apply buffered writes to the data store. Each discussion presents the two designs in turn.

A Change: A change is created by a write session. It may be idempotent or non-idempotent. Both designs must apply a non-idempotent write to the data store once. This is specially true with arbitrary failures of AppNodes. The definition of a change is specific to the data store's data model. It is different for the document data model (NoSQL) of Design 1 when compared with the relational data model (SQL) of Design 2. Below, we describe these in turn. Subsequently, we describe how each design supports non-idempotent changes.

With Design 1, a change by a write may be represented in a *JSON-like* format that is similar to MongoDB's update command. Table 2 shows examples of how these changes are

represented. In this table, `$set` is idempotent, while `$inc` is non-idempotent. A change that adds a value or an object to a set while guaranteeing uniqueness (e.g., `$addToSet` of MongoDB) is idempotent since it does not allow duplicates. However, a similar operation without uniqueness property (MongoDB's `$push`) is non-idempotent.

Document D	Write action applied to D	Idem-potent?	Representation of change	After applying the write to D
{ field: "val" }	Set value of *field* to *newval*	✓	{ "$set": { field: "newVal" } }	{ field: "newVal" }
{ field: "val" }	Remove a *field*	✓	{ "$unset": { field: "" } }	{ }
{ field: i }	Increment value of *field* by *x*	✗	{ "$inc": { field: x } }	{ field: i+x }
{ field: ["a"] }	Add to value of *field*	✓	{ "$addToSet": { field: "b" } }	{ field: ["a", "b"] }
{ field: ["a"] }	Add to value of *field*	✗	{ "$push": { field: "a" } }	{ field: ["a", "a"] }
{ field: ["a", "b"] }	Remove from value of *field*	✗	{ "$pull": { field: a } }	{ field: ["b"] }

Table 2: Example of write actions and the representation of their changes.

With Design 2, a change may be a SQL DML command: insert, delete, update. The command may impact multiple rows. Design 2 may represent the change as a string representation of the DML command issued by the application or a compact representation of it. In our TPC-C implementation, we use the latter to enhance utilization of both memory space and network bandwidth.

Designs 1 and 2 process non-idempotent changes in different ways to tolerate arbitrary failures of BGTs that apply these changes. Design 1 requires developers to provide additional software to convert non-idempotent changes to idempotent ones. BGTs replace a non-idempotent change with its equivalent idempotent change prior to applying it to the data store. Design 2 uses the transactional property of its data stores to apply non-idempotent changes only once. Its BGTs stores the id of a session applied to the data store in a special table/collection, *AppliedSessions*, as a part of the transaction that applies this session's changes to the data store. Prior to applying changes of a session to the data store, a BGT looks up the session id in the AppliedSessions table. If found then it discards the session and its buffered writes. Otherwise, it constructs a transaction consisting of the session's changes (SQL DML commands) along with the command that appends the id of the session to the AppliedSessions table. Next it executes this transaction and deletes the session object from the cache. Periodically, a BGT compacts the AppliedSessions table by deleting those session rows with no session objects in the CMI.

Buffered writes: A buffered write is a sequence of changes. With Design 1, atomicity is at the granularity of a document [29]. Hence, a buffered write represents a sequence of changes to one document. Each is a represented as a pinned cache entry, i.e., a key-value pair in a CMI. Buffered writes are partitioned across CMIs.

Definition 3.1. *With Design 1, a buffered write for a document D_i is represented as a key-value pair (k_i^{bw}, v_i^{bw}) where k_i^{bw} identifies a buffered write and v_i^{bw} stores either the final value of D_i or the pending changes to D_i. With the latter, the sequence of changes in v_i^{bw} represents the serial order of writes to D_i.*

The key k_i^{bw} may be constructed by concatenating "BW" with DocID where DocID is a unique identifier (or primary key) of the impacted document.

There are two approaches to buffer a change to v_i^{bw}: Append and Read-Modify-Write (RMW). Both acquire an X lease on the key k_i^{bw}. While Append requires the AppNode to

append its changes to v_i^{bw}, RMW requires the AppNode to read v_i^{bw}, update v_i^{bw} with the change, and write v_i^{bw} back to the cache. An efficient design of RMW grants an X lease as a part of read that fetches v_i^{bw}. RMW may compact v_i^{bw} by eliminating changes that nullify one another. Below, we present an example to illustrate these concepts.

Example 3.1. *Alice's document is impacted by two write actions: i) Bob invites Alice to be friend and ii) Alice accepts Bob's invitation. Representation of changes for i) is { "$addToSet": { pendingfriends: "Bob" } }, i.e., add Bob to Alice's pending friends. Representation of changes for ii) is { "$pull": { pendingfriends: "Bob" }, "$addToSet": { friends: "Bob" } }, i.e., remove Bob from pending friend list and add Bob to Alice's friend list. With Append, the buffered write is merely the JSON array that includes the two representations. With RMW, the buffered write becomes { "$addToSet": { friends: "Bob" } } since $pull cancels $addToSet of Bob to pendingfriends of Alice.* ■

In Example 3.1, both write actions make small changes to Alice's document. With append, the changes reflect the serial order of writes. The RMW performs a compaction to remove a redundant change.

Design 2 must support sessions that produce changes impacting rows of multiple tables. Thus, a buffered write represents a sequence of changes performed by a session. It is associated with a session object.

Definition 3.2. *With Design 2, a buffered write for a session T_i is represented as a key-value pair (k_i^{bw}, v_i^{bw}) where k_i^{bw} identifies the session T_i and v_i^{bw} stores either the raw SQL DML commands issued by that session or their compact representation. The sequence of changes in v_i^{bw} represents the serial order of SQL DML commands by session T_i.*

With both designs, buffered writes may be replicated across multiple CMIs to enhance their availability in the presence of CMI failures. These replicas are un-pinned and deleted after they are applied to the data store.

Mapping: A mapping implements a detective technique that provides read-after-write consistency when the application encounters misses for cache entries with pending buffered writes. It enables a cache miss to discover writes that must be used to compute the missing value.

An alternative to the detective approach of using mappings is to prevent cache misses for entries with pending buffered writes. The idea is to require a write session to generate the missing cache entry prior to generating a buffered write (to prevent a future cache miss). This solution must pin these cache entries to prevent their eviction until their buffered writes are applied to the data store. The challenge with this design is that it may pin entries that may not be referenced in the near future, reducing the cache hit rate of the application. Due to this limitation, we discard this preventive technique and assume use of mappings for the rest of this paper.

There are two ways to use a mapping. First, apply buffered writes prior to evicting a cache entry that depends on them. We term this the cache-server side (CSS) solution. Second, require the reads that observe a cache miss to look up the mapping to identify buffered writes and use them to compute the missing value. We term this technique the application-side solution (APS). APS may compute the missing value in different ways. In

its simplest, it may apply the buffered writes to the data store first, and query the data store to compute the missing value. Alternatively, if a buffered write provides sufficient information for the application to compute the missing value then it may do so, leaving the buffered write to be applied by a BGT.

With both APS and CSS, it is the responsibility of the developer to author software to specify mappings.

Definition 3.3. *A* mapping *inputs the key k_i for a cache entry to compute keys of its buffered writes, $\{k_i^{bw}\}$. With Design 1, these keys identify the documents with buffered writes that are used to compute the missing entry. With Design 2, these keys identify the session ids with pending buffered writes. A mapping may be a developer provided function or represented as a key-value pair (k_i^M, v_i^M).*

When a mapping is represented as a pinned key-value pair (k_i^M, v_i^M), k_i^M identifies mapping i uniquely. With Design 1 (2), its value v_i^M is the keys of those buffered writes (session objects) that impact k_i. A write that generates buffered writes must also generate a mapping. A read that observes a cache miss must always look-up $\{k_i^M\}$ in the cache and apply its identified buffered writes to the data store prior to querying it for the missing value. Many mappings for different cache entries may reside in a CMI.

Algorithm 1: Process a cache miss (Design 1).

```
1 function Process_Cache_Miss(k_i):
      // Use mapping to compute key of the buffered write
2     k_i^bw ← mapping(k_i);
3     if k_i^bw is null then
4         return; // Cache miss, no buffered write
5     Apply_Buffered_Write(k_i^bw);
6     Un-pin and delete explicit mappings (if any);
7     v_i ← Result of a function that queries the data store;
8     Insert (k_i, v_i) in CMI[hash(k_i)];
```

With APS, a read that observes a cache miss uses output of a mapping to look up the buffered write(s) in the CMI, see ② of Algorithm 1. If there is no mapping then it returns a cache miss so that the application may query the data store for the missing value ③-④. Otherwise, it applies the buffered writes to the data store prior to computing the missing cache entry ⑤.

Algorithm 2 applies a buffered write as an atomic session because, with Design 1, it must convert non-idempotent changes into idempotent ones. Algorithm 2's read of a buffered write for RMW obtains an X lease and fetches the buffered write ③. Next, it compacts the changes in this buffered write ④. If the buffered write contains one or more non-idempotent changes, it converts these changes to idempotent ones, writing a revised buffered write with idempotent changes only, ⑤-⑧. It calls itself recursively ⑨ to apply the idempotent writes to the data store, unpins and deletes the buffered write, and commits to release its X lease, ⑪-⑬.

Algorithm 2: Apply buffered write (Design 1).

```
1  function Apply_Buffered_Write(k_i^bw):
       // Look up buffered write
2      sessionId ← Generate a unique token;
3      v_i^bw ← Read(k_i^bw, sessionId); // Obtain a X lease on the buffered write
4      Compact v_i^bw and remove redundant changes;
5      if v_i^bw contains one non-idempotent change then
6          v_i^bw ← Idempotent equivalent of v_i^bw;
           // Replace value of buffered write with its idempotent equivalent
7          Set k_i^bw to v_i^bw;
8          Commit(sessionId); // Release X lease
9          Apply_Buffered_Write(k_i^bw);
10     else
11         Apply v_i^bw to document D_i in the data store;
           // Delete the buffered write
12         Un-pin and delete k_i^bw;
13         Commit(sessionId);
```

Similarly, CSS uses the output of the same mapping to locate buffered writes for a key that is being evicted, applies them (if any) to the data store per Algorithm 2. This technique incurs the overhead of an extra cache look up for every cache miss with APS and cache eviction with CSS.

APS and CSS are different in when and how they look up the mappings and process them. CSS uses the mappings when evicting a cache entry. APS uses the mappings when a read observes a cache miss. APS and CSS look up the mappings at the same rate with a workload that has the same rate of cache misses and evictions. If the cache replacement technique favors maintaining small sized cache entries by evicting large ones [17] then its rate of cache evictions would be lower than its cache misses. In this case CSS would do fewer look ups of mappings and processing of buffered writes.

There is also the complexity of implementing APS and CSS in a distributed manner. With an off-the-shelf cache manager such as memcached, it may be easier to implement APS instead of CSS. Without loss of generality, we assume APS for the rest of this paper.

Both Designs 1 and 2 may generate a Query Result Change, QRC. As suggested by its name, a QRC impacts the results of a query. A write that generates a buffered write *may* also generate a QRC. Algorithm 3 shows how a QRC is used to process a cache miss. The algorithm queries the data store ② and applies QRCs to its results to obtain the latest value ③-⑦. This final value is inserted in the cache for future look up. It is the responsibility of the application to maintain this entry up to date in the presence of writes that impact its value. Algorithm 3 does not delete QRCs. A BGT that applies a buffered write corresponding to one or more QRCs deletes them. This enables repeated applications of QRCs should a CMI evict the computed cache entry. We use Algorithm 3 to implement the Stock-Level transaction of TPC-C.

Discovery of buffered writes using PendingWrites and Queues: With both designs, a BGT must discover buffered writes and apply them to the data store. With Design 1, a special key termed *PendingWrites* identifies the buffered writes for different documents. A write session obtains an X lease on PendingWrites and appends its buffered write key (k_i^{bw}) to its value. To minimize contention among concurrent writes, we represent PendingWrites as α pinned sub-keys. Keys of the buffered writes (k_i^{bw}s) are hash partitioned across them. The value of α is typically a multiple of the number of concurrent writes (threads) and background threads, whichever is greater.

With Design 2, a queue maintains the serial order in which buffered writes of different sessions must be applied to the data store. It is represented as a key-value pair. The key identifies a queue known to a BGT and its value is a temporal order of session objects to be applied to the data store. (Each session object identifies a buffered write.) An application may maintain multiple queues. For example, with TPC-C, there is one queue per warehouse. When TPC-C is configured with W warehouses, it maintains W queues.

Algorithm 3: Process a cache miss using QRC.

```
1 function Process_Cache_Miss(k_i):
2     v_i ← Result of a function that queries the data store;
      // Use mapping to get a list of session ids
3     sessIds ← Get a list of session ids from mapping(k_i);
4     for each sessId in sessIds do
5         k_i^bw ← Construct the buffered write key from sessId;
6         v_i^bw ← Get buffered write k_i^bw;          // Obtain a S lease
7         Update v_i to include changes from v_i^bw;
8     Insert (k_i, v_i) in CMI;
```

Background Threads, BGTs: BGTs are threads that implement the asynchronous application of buffered writes to the data store. With Design 1, a BGT checks for dirty documents by looking up partitions of PendingWrites periodically. It processes each buffered write per Algorithm 2. This algorithm was presented in the context of processing a cache miss using a mapping.

Design 2 uses Algorithm 4 to apply a set of sessions to the data store. The set of sessions is obtained from a queue. For each session, it looks up the id of the session in the AppliedSessions table ⑤. If it exists then the session has already been applied to the data store and its session object is deleted. Otherwise, it creates a session and obtains the list of changes from each session object ⑪-⑬. It starts a data store transaction and inserts the id of the session in the AppliedSessions table ⑮. It merges the list of changes and applies them to the data store ⑯. Finally, it commits the data store transaction ⑰. It maintains the id of sessions in a set named sessIdsToDelete ⑳. Once this set reaches a pre-specified threshold β, Algorithm 4 deletes the session ids from the mappings, the queues, and the AppliedSession table.

Durability: Both designs pin their buffered writes, mappings, PendingWrites/Queues in

Algorithm 4: Apply sessions to the data store (Design 2).

```
1  function Apply_Buffered_Writes( sessIds ):
2      sessIdsToApply ← {};
3      sessIdsToDelete ← {};
4      for each sessId in sessIds do
5          if sessId exists in Applied-Sessions table then
6              Delete buffered write of sessId from the cache;
7              Delete sessId from the Applied-Sessions table;
8              continue;
9          Add sessId to sessIdsToApply;
10         if the list sessIds is exhausted or sessIdsToApply size reaches α then
               // Apply sessions as one transaction to the data store
11             sessionId ← Generate a unique token;
12             {k_i^bw} ← Construct keys from sessIdsToApply ;   // Obtain X leases on k_i^bw
13             {v_i^bw} ← multi_read({k_i^bw}, sessionId);
14             Start data store transaction;
15             Insert sessIdsToApply to Applied-Sessions table;
16             Merge {v_i^bw} and apply them to the data store;
17             Commit data store transaction;
18             multi_delete({k_i^bw}, sessionId);    // Delete buffered writes of sessIds from
                the cache
19             Commit(sessionId);                          // Release X leases on k_i^bw
20             Add ids from sessIdsToApply to sessIdsToDelete;
21             sessIdsToApply ← {};                                  // Clear the lists
               // Perform compaction
22             if the list sessIds is exhausted or sessIdsToDelete size reaches β then
23                 Delete sessIdsToDelete from the mappings;
24                 Delete sessIdsToDelete from the queue;
25                 Delete sessIdsToDelete from the Applied-Sessions table;
26                 sessIdsToDelete ← {};
```

a CMI to prevent its replacement technique from evicting them. Moreover, with multiple CMIs, both designs replicate these entries to enhance their availability in the presence of CMI failures.

4 Evaluation

This section evaluates the write-back policy using YCSB [9], BG [5], and TPC-C [42] benchmarks. All experiments use IQTwemcached [19] that implements S and X leases. With YCSB and BG, we use Design 1. TPC-C uses Design 2. While BG uses MongoDB [30] version 3.4.10 as its data store, both YCSB and TPC-C use MySQL version 5.7.23 as their

data store. We chose MongoDB and MySQL to highlight applicability of the write-back policy to both SQL and NoSQL data stores. Moreover, BG highlights the applicability of Design 1 to SQL when the workload is simple interactive social networking actions.

With all benchmarks, we quantify maximum memory used by the write-back policy assuming a sustained high system load. These maximums are unrealistic as a typical workload is diurnal consisting of both a low and a high load, e.g., see Facebook's load [4]. At the same time, it is useful to establish the worst case scenario.

Our experiments were conducted on a cluster of *emulab*[43] nodes. Each node has two 2.4 GHz 64-bit 8-Core (32 virtual cores) E5-2630 Haswell processors, 8.0 GT/s bus speed, 20 MB cache, 64 GB of RAM, connects to the network using 10 Gigabits networking card and runs Ubuntu OS version 16.04 (kernel 4.10.0). Unless stated otherwise, each experiment starts with a warm cache (100% cache hit rate) and runs for 10 minutes.

Obtained results highlight the following lessons:

1. Write-back enhances performance with all benchmarks as long as memory is abundant. It also enhances horizontal scalability as a function of the number of nodes in the caching layer. See Sections 4.1, 4.2, and 4.3.

2. With write-back, there is a tradeoff between the amount of required memory, the number of BGTs applying buffered writes to the data store, and the throughput observed by the application (foreground tasks). Increasing the number of BGTs reduces the amount of memory required by write-back. However, it also reduces the throughput observed by the application. See Sections 4.1 and 4.2.

3. Limited memory diminishes the performance gains provided by write-back. This is because it must pin buffered writes, mappings, PendingWrites/Queues in memory. These entries may reduce the cache hit rate observed by the application. One approach to mitigate this is to limit the amount of pinned memory allocated to write-back, forcing it to switch to write-through once this memory is exhausted. We observe memcached memory to become calcified [24], preventing write-back from using its assigned memory. See Section 4.4.1.

4. The overhead of replicating buffered writes, mappings, PendingWrites/Queues of write-back is in the form of network bandwidth. This overhead becomes negligible when writes are infrequent and the application's cache entries are much larger than these entries. Moreover, an increase in the number of nodes in the caching layer increases the available network bandwidth. This reduces the overall impact of replication. In our experiments, constructing 3 replicas with 4 cache servers reduces observed throughput with 1 replica by 19%. Experiments conducted with 8 cache servers observes a 6% decrease in throughput. See Section 4.4.2.

5. Our proposed write-back technique complements the write-back technique of both a data store such as MongoDB and a host-side cache such as Flashcache. Its enhances their performance more than 2x with workloads that exhibit a high read-write ratio. See Section 4.4.3.

Below, we present results in support of these lessons in turn.

4.1 YCSB: Design 1 with MySQL

The YCSB database consists of *10 million* records. Each record has 10 fields, each field is 100 bytes in size. A read or an update action reads or updates all fields of a record. A scan action (Workload S) retrieves 5 records (cardinality 5). When all data fits in IQTwemcached, the cache size for Workloads A, B and C is 14 GB. It is 55 GB with Workload S. We drop Workload C from further consideration because it is 100% read and a choice of a write-policy does not impact its performance. Unless stated otherwise, we assume a uniform access pattern. Details of the YCSB workloads are shown in Table 3.

Workload	**Actions**
Workload A	50% Read, 50% Update
Workload B	95% Read, 5% Update
Workload C	100% Read
Workload S	95% Scan, 5% Update

Table 3: YCSB Workloads.

Response time: Table 4 shows response time of YCSB Read, Scan and Update actions with MySQL by itself and alternative write policies. The numbers in parentheses are the number of round-trips from the application server to the data store and CMIs. The cache expedites processing of Reads and Scans because looking up results is faster than processing a query. Update is faster with write-back because it does not incur the overhead of generating log records. MySQL's response time is with Solid State Disk (SSD) as the mass storage device. With a Hard Disk Drive (HDD), MySQL's response time for Update increases to 35.87 milliseconds.

Write-through and write-around are slower than MySQL because they include the overhead of updating the cache. Write-around incurs the following 3 network round-trips: 1) acquire an X lease on the impacted cache entry from the CMI, 2) issue the SQL statement to MySQL, and 3) commit the session. Write-through updates the impacted cache entry using a read-modify-write[2], increasing the number of network round-trips to 4. Write-back increases the number of network round-trips to 5 by writing its buffered write to the CMI[3].

Throughput: Figure 1 used as motivation in Section 1 shows the throughput with the

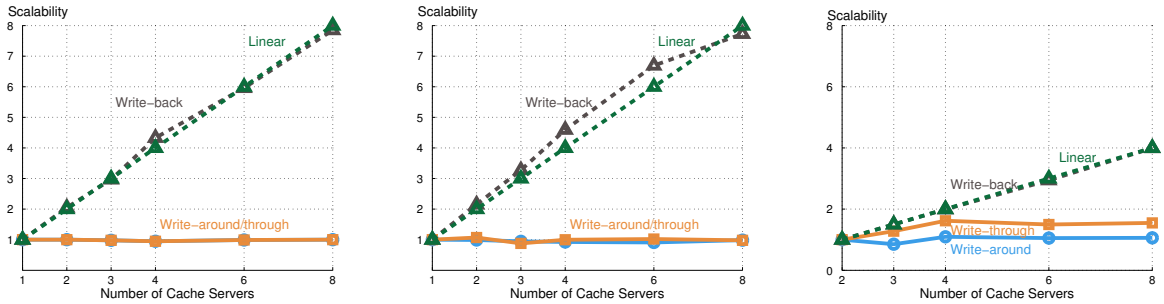
[2]Read of RMW obtains an X lease on the referenced key in addition to reading its value.

[3]Write-back's 5 network round-trips are for as follows. Two round-trips for the RMW of the cache entry. Two round-trips for the RMW of the buffered write. One round-trip for commit that releases the X leases and makes the cache writes permanent.

Action	**MySQL**	**MySQL + Write-Back**	**MySQL + Write-Through**	**MySQL + Write-Around**
Read	0.20 (1)	0.08 (1)		
Scan	0.36 (1)	0.15 (1)		
Update	0.55 (1)	0.44 (5)	0.92 (4)	0.88 (3)

Table 4: Response time (in milliseconds) of YCSB actions.

alternative YCSB workloads using write-back, write-around, and write through as a function of cache servers. In these experiments, we prevent the global LRU lock of IQTwemcached from limiting performance by launching 8 instances of cache manager (CMI) per server. Each CMI is assigned 8 GB of memory. We increase the number of servers (CMIs) from 1 (8) to 8 (64). Obtained results show that write-back outperforms its alternatives[4] by a wide margin.



(a) Workload A (b) Workload B (c) Workload S

Figure 4: Scalability with different YCSB workloads.

Figure 4 shows scalability of Workloads A, B, and S. These correspond to the throughput numbers of Figure 1. With Workloads A and B, scalability is computed relative to 1 cache server. Write-around and write-through do not scale because either the data store CPU or disk/SSD becomes the bottleneck. Write-back scales linearly by buffering writes and eliminating the data store from the processing path. While the bottleneck resource is CPU with Workload A, the bottleneck resource is network bandwidth with Workloads B and S.

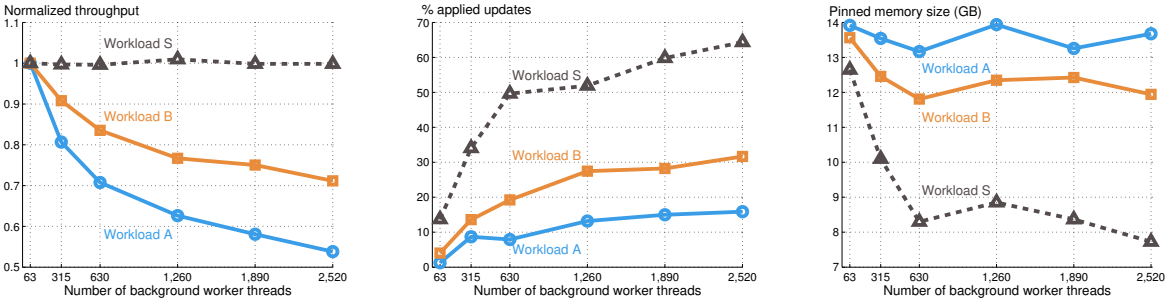
With Workload S, we show scalability relative to the configuration consisting of 2 cache servers. Its larger cache entries prevent it from observing a 100% cache hit with 1 cache server (and observes a 100% cache hit with 2 or more cache servers). Using 1 cache server as the basis results in a super-linear scaleup. Using 2 servers as the basis of the scalability graph provides for a more subjective evaluation.

4.1.1 Required Memory

The rate at which a system applies buffered writes to the data store is a tradeoff between the cache memory space and decrease in rate of processing (throughput) observed by the foreground requests. The foreground requests are impacted for two reasons. First, background threads compete with foreground threads that observe cache misses for using the data store. These foreground threads must apply their changes to the data store and query it to compute the missing cache entries. Second, application of buffered writes requires network bandwidth for (background threads of) AppNodes to fetch the buffered writes from CMIs. At the same time, an aggressive application of buffered writes deletes both the buffered writes and their mappings from CMIs faster, minimizing the amount of memory required by write-back.

We demonstrate the tradeoff with an experiment that consists of 63 instances of AppNodes hosted on 21 servers (3 AppNode instance per server), and 64 CMIs hosted on 8 servers

[4]Throughput of MySQL by itself is 17,523, 111,156, and 91,499 for Workloads A, B, and S, respectively.



(a) Throughput relative to 63 BGTs

(b) % Applied updates

(c) Pinned memory size

Figure 5: Impact of the number of background workers on throughput, percentage of updates applied to the data store, and the total amount of pinned memory (YCSB workloads).

(8 CMIs per server). We vary the number of background threads (BGTs) from 1 per AppNode instance to 5, 10, 20, 30, and 40. Each BGT applies buffered writes to the data store as fast as possible.

Figure 5a shows the normalized throughput observed with different number of BGTs relative to 1 BGT per AppNode instance (63 BGTs). The x-axis of this figure is the total number of BGTs. Increasing the number of BGTs decreases throughput of write-heavy workload A (47% drop with 2,520 BGTs) followed by read-heavy Workload B (30% drop with 2,520 BGTs). Workload S observes a negligible decrease in its throughput even though it is a read-heavy workload similar to B. Moreover, the size of buffered writes with both B and S are identical. S is different because it consists of scans with cache entries that are 5 times larger than those of B. The network bandwidth becomes the bottleneck with S to render the overhead of additional BGTs negligible.

Figure 5b shows the percentage of changes applied by the background threads to the data store at the end of an experiment. Workload A has the lowest percentage because it is write-heavy. However, this percentage increases modestly ($< 20\%$) as we increase the number of BGTs. This explains why Workload A has the highest amount of pinned memory in Figure 5c.

Figure 5 shows Workload S benefits the most from an increase in the number of BGTs. This is because its normalized throughput is comparable to having 63 BGTs while its percentage of applied writes to the data store increases dramatically. This in turn reduces its pinned memory size by almost 2x.

4.2 BG: Design 1 with MongoDB

BG [5] is a benchmark that emulates interactive social networking actions. It quantifies Social Action Rating (SoAR) defined as the highest throughput that satisfies a pre-specified service level agreement (SLA). The SLA used in our experiments is 95% of actions processed in 100 milliseconds or faster. In our evaluation, we use a social graph consisting of 10 million users with 100 friends per user. We considered three workloads as shown in Table 5. List friends (or list pending friends) action only returns 10 out of 100 friends (or pending friends) of the requested user.

BG Social Actions	90% reads	99% reads	99.9% reads
View Profile	80%	89%	89.9%
List Friends	5%	5%	5%
List Pending Friends	5%	5%	5%
Invite Friend	4%	0.4%	0.04%
Reject Friend	2%	0.2%	0.02%
Accept Friend	2%	0.2%	0.02%
Thaw Friendship	2%	0.2%	0.02%

Table 5: BG Workloads.

Response time: Table 6 shows the response time of BG's actions with MongoDB by itself and with IQTwemcached using write-back, write through, and write-around. MongoDB was configured to acknowledge writes immediately after writing it in its memory, flushing dirty documents to its mass storage device very 60 seconds (writeConcern=ACKNOWLEDGED). These numbers were gathered using a single threaded BG generating requests.

While the cache improves the response time of the read actions with alternative write policies, its response time for the write actions is worse than MongoDB by itself. This is due to a higher number of round-trips to the cache, see the numbers in parentheses. If MongoDB was configured to generate log records and flush them to disk prior to acknowledging the write (writeConcern=JOURNALED), write-back would have been faster [5]

Action	MongoDB	MongoDB + Write-Back	MongoDB + Write-Through	MongoDB + Write-Around
View Profile	0.52 (1)	0.18 (1)		
List Friends / Pending Friends	0.81 (1)	0.52 (2)		
Invite Friend	0.23 (1)	0.34 (5)	0.70 (5)	0.62 (4)
Reject Friend	0.24 (1)	0.41 (6)	0.71 (6)	0.66 (4)
Accept Friend	0.46 (2)	1.01 (11)	1.64 (11)	1.41 (6)
Thaw Friendship	0.48 (2)	0.87 (11)	1.66 (11)	1.21 (5)

Table 6: Response time (in milliseconds) of BG actions with MongoDB in acknowledged mode, writeConcern=ACKNOWLEDGED. Number of network round-trips is shown in parentheses.

Write-back performs the write actions faster than write-through and write-around because it performs the write to MongoDB asynchronously. Write-around outperforms write-through by reducing the number of round-trip messages because deleting impacted cache entries requires 1 round-trip versus 2 round-trips to update a cache entry using RMW with write-through.

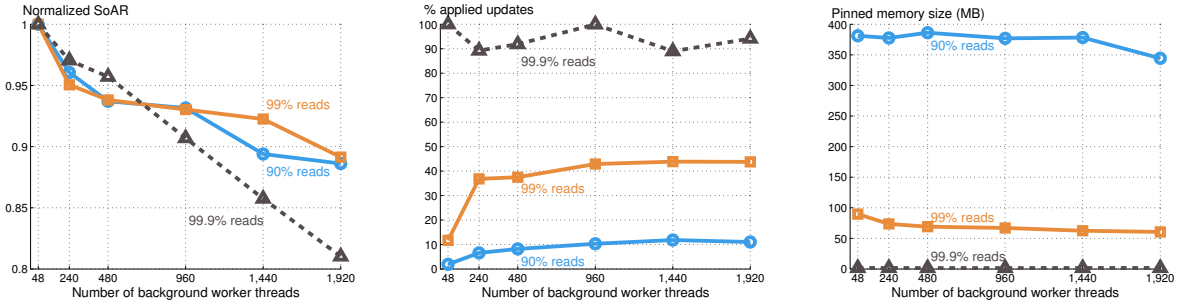
[5]MongoDB with writeConcern=JOURNALED provides the following response time: 0.6 milliseconds for each of Invite Friend and Reject Friend, and 1.2 milliseconds for each of Accept Friend and Thaw Friendship.

One may use Table 6 in combination with the frequencies shown in Table 5 to compute the average response time. With BG's lowest read-write ratio of 90%, the average response time with write-back (0.26 msec) is faster than MongoDB by itself (0.53 msec), with write-through (0.30 msec) and write-around (0.29 msec). Alternative write policies are faster than MongoDB because the View Profile action dominates (89%) the workload and the cache improves its performance dramatically. With BG's high read-write ratio of 99.9%, write-back becomes significantly faster than MongoDB by itself. Moreover, write-through and write-around provide a response time comparable to write-back.

Workload	**SoAR (actions/sec)**		
	MongoDB	Write-back	Write-through
90% read	27,518	355,769	111,548
99% read	48,365	693,816	488,593
99.9% read	76,068	711,659	678,665

Table 7: SoAR of MongoDB by itself and with 1 cache server.

Throughput: Write-back provides a higher SoAR when compared with other policies, see Table 7. Its SoAR is dictated by the network bandwidth of the cache server. Moreover, it enables all BG workloads to scale linearly as a function of cache servers. Write-around and write-through scale sub-linearly by no more than a factor of 6 with 8 cache servers. Scalability is lowest with 90% reads because the CPU of MongoDB becomes fully utilized processing writes. Scalability improves with 99% and 99.9% reads as the load on MongoDB is reduced and the network card of the cache servers becomes fully utilized.



(a) SoAR relative to 48 BGTs (b) % Applied updates (c) Pinned memory size

Figure 6: Impact of the number of background workers on SoAR, percentage of updates applied to the data store, and the total amount of pinned memory (BG workloads).

4.2.1 Required Memory

Similar to YCSB results, the amount of memory required for buffered writes with BG is a function of their production rate by the foreground threads and application to the data store by BGTs. Obtained results are shown in Figure 6. A key difference is that with its read-heavy (99.9%) workload, the BGTs apply buffered writes to the data store at the same rate at which writes produce them. The measured SOAR is impacted by the overhead of BGTs

Transaction	MySQL	MySQL+Write-Back	MySQL+Write-Through
New-Order	8.06	5.18	10.99
Payment	2.09	1.27	3.29
Order-Status	1.59	0.71	
Delivery	22.36	3.56	24.09
Stock-Level	2.00	1.35	

Table 8: Response time (in milliseconds) of TPCC actions.

checking for buffered writes to find none. This is because the network is the bottleneck. Hence, increasing the number of BGTs reduces throughput without providing a benefit, see Figure 6a.

4.3 TPC-C: Design 2 with MySQL

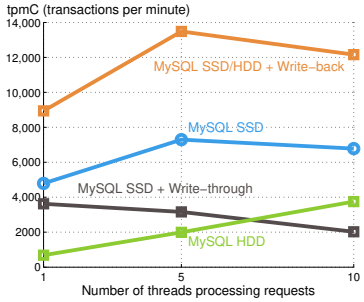
TPC Benchmark C [42] is an on-line transaction processing (OLTP) benchmark. TPC-C models a supplier operating out of several warehouses and their associated sales districts. It quantifies transactions per minute (tpmC) rating of a solution. We use the standard setting of TPC-C with 1 warehouse consisting of ten districts per warehouse, each district serving three thousand customers, and 100,000 items per warehouse. These results are obtained using OLTP-Bench [14] implementation of TPC-C extended with leases for strong consistency. The deployment consists of 2 emulab nodes. One hosting the OLTP-Bench workload generator and the other hosting (a) MySQL by itself and (b) MySQL and 8 instances of IQTwemcached using either write-through or write-back policies.

We analyze MySQL configured with either solid state drive (SSD) or hard disk drive (HDD). Write-back improves performance throughput of MySQL with SSD by more than two folds. Moreover, its performance is not sensitive to whether MySQL is configured with either SSD or HDD because TPC-C's database is small, causing cache to observe a 100% hit rate [6].

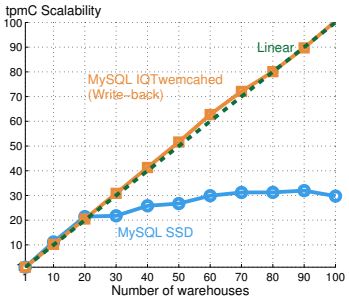
Response time: Table 8 shows the average response time of each TPC-C transaction using MySQL by itself, with write-back, and write-through. Write-back improves the response time of all transactions. Write-through improves the response time of read-only transactions: Order-Status and Stock-Level. However, write-through causes the write transactions to incur the overhead of updating both the cache entries and MySQL. Hence, it is slower than MySQL by itself and with write-back.

The frequency of New-Order, Payment, Order-Status, Delivery, and Stock-Level transactions are 45%, 43%, 4%, 4%, and 4%. The weighted response time with MySQL is 5.56 milliseconds. Write-back is faster at 3.10 milliseconds, providing a 44% enhancement. Write-through is 33% slower than MySQL because the overhead of writes outweighs the benefits observed by the reads that constitute 8% of the workload.

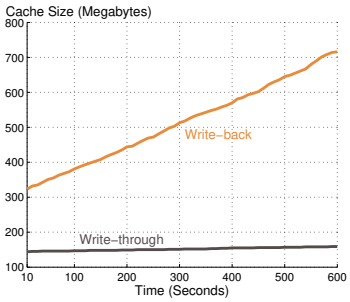
[6] MySQL with HDD slows down the rate at which BGTs apply buffered writes, requiring more memory than MySQL with SSD.



(a) tpmC of MySQL by itself and with IQTwemcached configured with write-back and write-through.



(b) tpmC Scalability.



(c) Memory size of IQTwemcached with write-back and write-through, 5 TPC-C threads, 1 warehouse.

Figure 7: tpmC of MySQL by itself and with IQTwemcached configured with write-back and write-through.

Throughput: Figure 7a shows the write-through policy is inferior to MySQL by itself because 92% of TPC-C transactions are writes. Write-through must apply these writes to both MySQL and IQ-Twemcached synchronously. The overhead of writing to IQ-Twemcached outweighs benefits provided by its cache hits.

In Figure 7a, tpmC of write-back levels off with 5 and more threads due to contention for leases between the foreground threads. These foreground threads also contend with the background thread for an X lease on the queue of sessions. The same applies to MySQL with its lock manager blocking transactions to wait for one another.

Figure 7b shows scalability of TPC-C with MySQL and write-back as we increase the number of warehouses from 1 to 100 with 1 thread issuing transactions to one warehouse. MySQL's SSD becomes fully utilized with more than 20 threads (warehouses), limiting its scalability. Write-back scales linearly with network bandwidth of the caching tier dictating its scalability.

4.3.1 Required Memory

Figure 7c shows the amount of required memory with write-through and write-back policies with 1 warehouse. These results highlights the amount of memory required by the cache entries for TPC-C. Its increase with write-through as a function of time highlights the growing size of database (new orders) and its corresponding cache entries. These entries are included in the memory size reported for write-back. Moreover, write-back includes buffered writes, mappings, and queues. The difference between write-back and write-through highlights (a) the extra memory required by write-back for buffered writes and (b) faster processing of new orders increases both database size and cache size at a faster rate. Write-back requires a significantly higher amount of memory than write-through.

4.4 Discussion

This section discusses impact of limited memory on write-back, overhead of replicating buffered writes for durability, and tradeoffs associated with deploying write-back at different software layers.

4.4.1 Limited Memory and Slab Calcification

Reported performance numbers assume abundant amount of memory. Write-back performance degrades considerably with limited memory because (a) buffered writes compete with cache entries for memory to increase the cache miss rate observed by the application, (b) cache misses require buffered writes to be applied to the data store in a synchronous manner that diminishes the performance of write-back to be comparable to write-though[7], (c) cache managers such as memcached may suffer from slab calcification when memory is limited. We highlight these using YCSB.

This section illustrates the impact of limited memory with Design 1 using YCSB workload. We assume 16 AppNodes each with 10 BGTs, 1 MongoDB[8] server, 1 cache server with 8 CMI instances and a total of 14 GB of memory. This 14 GB cache space is enough for YCSB Workloads A and B to observe a 100% cache hit rate with the write-through policy. With the write-back policy, both the buffered writes and mappings start to compete with the cache entries for space. We vary the maximum amount of memory that can be occupied by these pinned key-value pairs to quantify the impact of limited memory on write-back when compared with write-through.

Figure 8 shows the throughput of Workload A increases as we increase the size of pinned memory from 2 to 12 GB. Logically, an increase in the value of x-axis leaves a lower amount of memory for regular key-value pairs to observe cache hits. Consider results shown for Workloads A and B in turn.

The throughput of Workload A is 52K with write-through and independent of the x-axis values. With 2 GB of pinned memory, write-back exhausts this memory with buffered writes quickly. Every subsequent write generates its buffered write for the cache only to be rejected, causing the write to update the data store synchronously (switches to write-through). Repeated failed attempts to insert buffered writes lowers write-back throughput, enabling write-through to outperform it. As we increase the size of pinned memory, a larger fraction of writes are performed using write-back, enabling write-back to outperform write-through. However, this reduces cache hit rate from 87% with 2 GB to 48% with 12 GB of pinned memory due to smaller memory size for regular key-value pairs that service read requests. Hence, the throughput observed with write-back levels-off, outperforming write-through by 2x. (If the memory was increased to 48 GB then write-back throughput would have been 6x higher.)

With 12 GB of pinned memory, write-back pins only 10 GB of data for Workload A. This is due to slab calcification [24, 6, 8]. To elaborate, memcached constructs two slab classes: Class 1 stores mappings. Class 2 stores regular key-value pairs and the buffered

[7]Figure 7a shows write-through to be inferior to MySQL by itself with TPC-C.

[8]Our switch from MySQL of Section 4.1 with YCSB to MongoDB is intentional to highlight flexibility of Design 1 to work with alternative data models.

writes as they are approximately the same size[9]. Once memory is assigned in its entirety, these two slab classes are calcified and may not allocate memory from one another. Once memory space assigned to Class 1 is exhausted, no additional mappings may be inserted in the cache. This means a subsequent write must be processed synchronously even though there is space[10] in Class 2 for buffered writes.

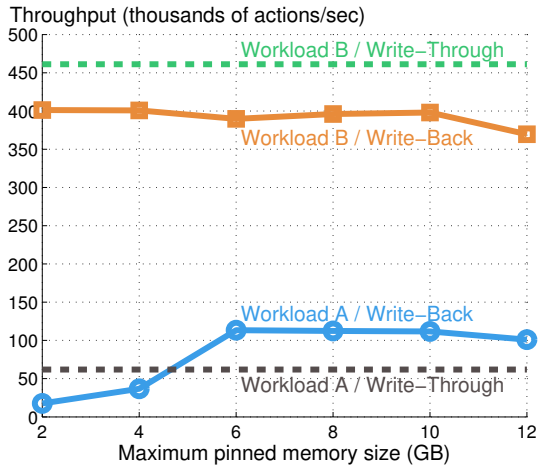


Figure 8: Impact of limited memory on write-back performance. Cache memory size is 14 GB. YCSB Zipfian constant is 0.75.

The slab calcification phenomena is magnified with Workload B, causing write-back to provide a lower performance than write-through for the entire spectrum of maximum pinned memory sizes. Only 5% of Workload B is writes, causing the slab class for pinned mappings to be significantly smaller. Thus, only 5.6 GB of memory can be used for buffered writes even though maximum pinned memory size is significantly larger, i.e., x-axis values of 6, 8, 10 and 12 GB.

The degree of skew in access pattern to data items is beneficial to write-back and impacts the observed trends. It increases the likelihood of several writes impacting the same MongoDB document. The write-back policy merges their changes into one pinned buffered write. Moreover, the number of mappings is reduced. Both minimize the number of pinned key-value pairs. In turn this results in a higher cache hit[11] rate, enhancing throughput by 40-50% with both workloads and a very skewed access pattern. Now, write-back outperforms write-through with both workloads.

A future research direction is to compare write-back with write-through using cache managers that do not suffer from slab calcification. Another research direction is to extend write-back to a dynamic framework that detects limited memory and switches to write-through, preventing a degradation in system performance. Alternatively, an algorithm may adjust the size of pinned memory (buffered writes and mappings) to maximize throughput.

4.4.2 Replication of Buffered Writes

Replication of buffered writes enhances their availability in the presence of cache server failures. At the same time, it consumes network bandwidth to transmit these redundant replicas to CMIs and the additional memory required to store them. While only one replica is fetched by a BGT to apply to the data store, the BGT must delete all replicas. This overhead impacts system throughput. This section quantifies this overhead by comparing the throughput of the write-back configured with 1 and 3 replicas for buffered writes, their mappings, and PendingWrites/Queues.

[9] A write updates all properties of a document.

[10] This space does not sit idle and is occupied by the application's cache entries.

[11] Workload B observes a 10% increase in cache hit rate.

Obtained results highlight the following lesson. The overhead of constructing 3 replicas becomes less significant as we 1) increase the size of the caching layer, 2) reduce the frequency of writes and 3) have workloads with cache entries much larger than the buffered writes and their mappings.

Figure 9 highlights the above lessons by showing the throughput of YCSB workload B with 4 and 8 cache servers using Design 1. (Each cache server hosts 8 CMIs.) The overhead of constructing 3 replicas with 4 cache servers lowers throughput by 19%. It is reduced to 6% with 8 cache servers. The larger configuration has a higher network bandwidth, reducing the overhead of replication more than three folds.

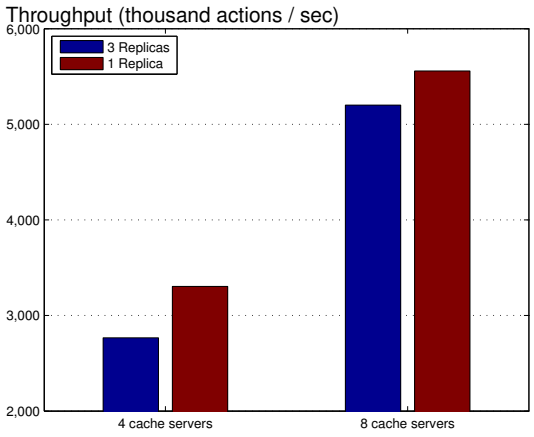


Figure 9: Impact of replicating buffered writes on throughput of YCSB Workload B.

With Workload S, the impact of replicating buffered writes is not noticeable with both configurations. This is because writes are 5% of the workload and the size of buffered writes and their mappings is insignificant relative to cache entry sizes. The network bandwidth limits the throughput of this workload with both 1 and 3 replicas.

4.4.3 Comparison with Alternative Write-Back Caches

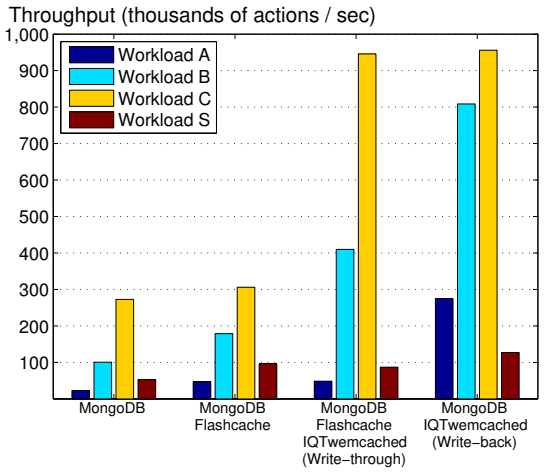


Figure 10: Performance of alternative cache configurations with YCSB. MongoDB is configured with writeConcern set to ACKNOWLEDGED.

With CADS architecture, one may apply the write-back policy in different software layers. For example, MongoDB implements write-back by acknowledging a write as soon as it is stored in its buffers (writeConcern is set to ACKNOWLEDGED). It flushes buffered writes to disk every 60 seconds. These buffered writes are not durable and if MongoDB crashes then they are lost.

Host-side caches stage disk pages referenced by a data store such as MongoDB in SSD to enhance performance. They are typically deployed seamlessly using a storage stack middleware or the operating system [11, 13, 37, 28, 7, 38, 21, 26, 25]. Examples include Flashcache [28] and bcache [37]. One may configure a host-side cache with alternative write-policies.

While these caches complement the application-side cache [1] and its write-back policy, a key question is how do they compare with one another? This section shows write-back using the application-side cache outper-

forms the other alternative by several orders of magnitude with both YCSB and BG. This is true even when the other two types of caches are combined together. Below, we present results using YCSB. BG provides similar observations.

Figure 10 compares the performance of four cache configurations using four YCSB workloads. The alternative configurations include: MongoDB, MongoDB with Flashcache, MongoDB with Flashcache and IQTwemcached using write-through, and MongoDB with IQTwemcached using write-back. In all configurations, MongoDB is configured with writeConcern set to ACKNOWLEDGED. The four YCSB workloads are shown in Table 3. Results are obtained using 1 server for MongoDB and its Flashcache, 1 server for IQTwemcached, and 8 AppNode servers generating requests. Consider results of each workload in turn.

YCSB Workload C is 100% read and Figure 10 highlights the benefits of using an application-side cache when compared with MongoDB either by itself or Flashcache. Application-side cache enhances throughput more than 3 folds regardless of the write-policy. The cache provides for result look up instead of MongoDB processing a query, improving throughput dramatically.

Workload B benefits from the application-side cache because 95% of its requests are identical to Workload C. The remaining 5% are writes that benefit from the write-back policy, enabling it to out-perform write-through almost 2x.

YCSB Workload A is write-heavy with 50% update. MongoDB performance with Flashcache is enhanced 2 folds because writes with SSD are faster than HDD. Using Linux fio benchmark, we observe the SSD IOPS for 4K block size to be 1.14x higher than HDD for sequential reads, 1.17x for sequential writes, 150x for random reads, and 50x for random writes. Every time MongoDB flushes pending writes to disk using fsync, it blocks write operations until fsync completes. Using SSD instead of HDD expedites fsync to improve performance.

Workload A does not benefit from an application-side cache configured with the write-through policy. However, its throughput is improved more than 5x with the write-back policy because writes are buffered in IQTwemcached and applied to MongoDB asynchronously.

Workload S utilizes network bandwidth of the cache server fully with both write-through and write-back policies. The improvement it observes from using write-back is 30% because 5% of its requests are writes. It is interesting to note that MongoDB with Flashcache provides a comparable throughput to the write-through policy, rendering the application-side ineffective for this workload. The explanation for this is that the network bandwidth of MongoDB with Flashcache becomes fully utilized with Workload S, enabling it to provide a comparable throughput.

5 Related Work

Write-back policy has been studied extensively in the context of host-side caches that stage disk blocks onto flash to enhance system performance [7, 13, 21, 25]. They are different than the application-side caches in several ways. First, application-side caches use DRAM that is both faster and provides lower capacities than NAND Flash assumed by host-side caches. Second, while host-side caches are transparent, application-side caches are non-transparent. The latter requires custom code by a software developer to cache arbitrary

sized objects (not fix-sized blocks of data), represent changes by a write, generate mappings, and maintain PendingWrites/Queues. Third, application-side caches have no concept of a dirty block that must be written to disk prior to being evicted. Hence, we pin buffered writes to prevent their eviction. Finally, while writes with host-side caches are idempotent, writes with application-side caches may either be idempotent or non-idempotent.

Write-back is a common feature of distributed caches and caching middleware such as EhCache [40], Oracle Coherence [33], Infinispan [23] or IBM WebSphere eXtreme Scale [22]. Coherence, Ignite, and EhCache are similar[12] and we describe them collectively using Coherence's terminology. Subsequently, we present IBM WebSphere Scale.

Coherence provides a simple "put(key,value)" method that (1) inserts the key-value pair in the cache, overwriting it if it exists, and (2) places the key-value pair in a CacheStore queue before returning success. Coherence is data store agnostic by requiring the developer to implement the "store()" interface of CacheStore. After a configurable time interval, a background thread invokes store() to persist the (key,value) pair to the data store. A read, issued using get(key), either observes the latest value from the cache or requires the cache to load the missing cache entry. This is realized by requiring the developer to implement the "load()" interface that queries the data store for the missing key-value pair. While the Coherence documentation is not specific about the details of how cache misses are processed, we speculate their processing considers the queued writes to provide read-after-write consistency.

WebSphere eXtreme Scale cache implements maps. A map is a collection of cache entries comparable to a CMI instance. An application may configure a loader (similar to CacheStore interfaces of Coherence) for a map. With write-back, it creates a thread to process delegating requests coming to a loader. When a write inserts, updates or deletes an entry from a map, a LogElement object is generated and queued. Each LogElement object records the operation type (insert, update or delete) and the new and old values. Each map has its own queue. A write-behind thread is initialized to periodically remove a queue element and apply it to the data store. LogElement objects are similar to our idempotent buffered writes using Append approach. However, the application of these writes to a SQL data store may violate referential integrity constraints. This is because data updated to different maps in one session are applied to the data store as different transactions. If there is a foreign key dependency between them, it is possible for an out of order write that violates this dependency. WebSphere recommends the data store to not have such constraints and allow out of order application of writes.

Our proposed write-back is novel in several ways. First, it supports non-idempotent changes. Coherence specifies changes must be idempotent. Same is true with a LogElement of WebSphere with the new and old values. Second, both Design 1 and 2 require mappings to implement read-after-write consistency in the presence of cache misses. All other caches lack this concept. Third, with Design 2, a session's changes to the data store are maintained in a session object. These changes are applied as a transaction to the data store, preserving the referential integrity constraints that exists between updates of a single session. Design

[12]There are subtle differences between these caches. For example, in a cluster deployment, EhCache may not apply pending writes in the order written to the cache. Infinispan, Coherence, and Ignite use a queue similar to our Design 2 to apply data store writes of different sessions based on their commit time.

2's queue ensures sessions are applied to the data store in the same serial order as their commit order using S and X leases. Finally, there are minor architectural differences. For example, WebSphere assumes the background threads are co-located with maps (CMIs) and execute independently. We assume these threads are co-located with AppNodes and use the concept of session with leases to prevent undesirable race conditions.

Systems such as Everest [15] or TARDIS [16] use "partial write-back". They buffer writes under special conditions. Everest improves the performance of overloaded volumes. Each Everest client has a base volume and a store set. When the base volume is overloaded, the client off-loads the writes to the idle stores. When the base volume load fall below a threshold, the client uses background threads to apply writes to base volume. TARDIS buffers writes in the cache when the data store is unavailable (either because the data store server has failed or a network partition prevents the application server from communicating with the data store). Subsequently, when the data store is available, TARDIS enters the recovery mode. During this mode, the application server retrieves the buffered writes from the cache and applies them to the data store. With our write-back design, background worker threads apply the buffered writes every time they are present in the cache. This is different than partial write-back technique where worker threads apply buffered writes during recovery mode only.

6 Conclusions and Future Work

Write-back is more complex to design and implement than write-through and write-around. However, with abundant amount of memory, it enhances system performance dramatically by enabling the caching layer to absorb writes and dictate the overall system performance. Write-back is in synergy with today's dropping price of DRAM and enterprise servers with more than half a terabyte of DRAM.

Our future research directions are as follows. First, we are analyzing the impact of a skewed pattern and its relevant load balancing techniques on write-back. Second, we are investigating persistent caches that use SSD [12, 39] to minimize cost of storing buffered writes. Third, we are developing models to quantify dollar benefits of using write-back instead of scaling the data store layer. These models capture cost of memory required by write-back and savings in the form of fewer servers, smaller rack-space footprint, and energy efficiency. Fourth, we are investigating auto-tuning algorithms that detect when memory is limited to switch from write-back to write-through and vice-versa. Finally, we are analyzing whether write-back caches enhance performance of in-memory data stores [27].

7 Acknowledgments

We gratefully acknowledge use of Utah Emulab network testbed [43] ("Emulab") for all experimental results presented in this paper.

References

[1] Y. Alabdulkarim, M. Almaymoni, Z. Cao, S. Ghandeharizadeh, H. Nguyen, and L. Song. A Comparison of Flashcache with IQ-Twemcached. In *IEEE CloudDM*, 2016.

[2] S. Apart. Memcached Specification, http://code.sixapart.com/svn/memcached/trunk/server /doc/protocol.txt.

[3] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. K. III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[5] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *CIDR*, January 2013.

[6] A. Blankstein, S. Sen, and M. J. Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *USENIX ATC*, 2017.

[7] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side Flash Caching for the Data Center. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.

[8] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *NSDI*, pages 379–392, 2016.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Cloud Computing*, 2010.

[10] Couchbase. Couchbase 2.0 Beta, http://www.couchbase.com/.

[11] S. Daniel and S. Jafri. Using NetApp Flash Cache (PAM II) in Online Transaction Processing. *NetApp White Paper*, 2009.

[12] B. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-value Store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, September 2010.

[13] DELL. Dell Fluid Cache for Storage Area Networks, http://www.dell.com/learn/us/en/04/solutions/fluid-cache-san, 2014.

[14] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB*, 7(4):277–288, 2013.

[15] R. Draves and R. van Renesse, editors. *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 2008.

[16] S. Ghandeharizadeh, H. Huang, and H. Nguyen. Teleporting Failed Writes with Cache Augmented Data Stores. In *Cloud*, 2018.

[17] S. Ghandeharizadeh, S. Irani, J. Lam, and J. Yap. CAMP: A Cost Adaptive Multi-Queue Eviction Policy for Key-Value Stores. *Middleware*, 2014.

[18] S. Ghandeharizadeh and J. Yap. Cache Augmented Database Management Systems. In *ACM SIGMOD DBSocial Workshop*, June 2013.

[19] S. Ghandeharizadeh, J. Yap, and H. Nguyen. IQ-Twemcached. http://dblab.usc.edu/users/IQ/.

[20] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, pages 677–680. Morgan Kaufmann, 1993.

[21] D. A. Holland, E. L. Angelino, G. Wald, and M. I. Seltzer. Flash Caching on the Storage Client. In *USENIX ATC'13 Proceedings of the 2013 USENIX conference on Annual Technical Conference*. USENIX Association, 2013.

[22] IBM. WebSphere eXtreme Scale, https://goo.gl/smgC3W.

[23] Infinispan. Infinispan, http://infinispan.org/.

[24] S. Irani, J. Lam, and S. Ghandeharizadeh. Cache Replacement with Memory Allocation. *ALENEX*, 2015.

[25] H. Kim, I. Koltsidas, N. Ioannou, S. Seshadri, P. Muench, C. Dickey, and L. Chiu. Flash-Conscious Cache Population for Enterprise Database Workloads. In *Fifth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2014.

[26] D. Liu, N. Mi, J. Tai, X. Zhu, and J. Lo. VFRM: Flash Resource Manager in VMWare ESX Server. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–7. IEEE, 2014.

[27] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 604–615, 2014.

[28] D. Mituzas. Flashcache at Facebook: From 2010 to 2013 and Beyond, https://www.facebook.com/notes/facebook-engineering/flashcache-at-facebook-from-2010-to-2013-and-beyond/10151725297413920, 2010.

[29] Mongo Inc. Mongo Atomicity and Transactions, https://docs.mongodb.com/manual/core/write-operations-atomicity/.

[30] MongoDB Inc. MongoDB, https://www.mongodb.com/.

[31] MySQL. Designing and Implementing Scalable Applications with Memcached and MySQL, A MySQL White Paper, June 2008, http://www.mysql.com/why-mysql/memcached/.

[32] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, pages 385–398, Berkeley, CA, 2013. USENIX.

[33] Oracle. Oracle Coherence, http://www.oracle.com/technetwork/middleware/coherence.

[34] Oracle Corporation. Oracle Database 18c, https://docs.oracle.com/en/database/oracle/oracle-database/18/index.html.

[35] RedisLabs. Redis, https://redis.io/.

[36] A. Sainio. NVDIMM: Changes are Here So Whats Next. *In-Memory Computing Summit*, 2016.

[37] W. Stearns and K. Overstreet. Bcache: Caching Beyond Just RAM. https://lwn.net/Articles/394672/,http://bcache.evilpiepirate.org/, 2010.

[38] STEC. EnhanceIO SSD Caching Software, https://github.com/stec-inc/EnhanceIO, 2012.

[39] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced Photo Caching on Flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, Santa Clara, CA, 2015.

[40] Terracotta. Ehcache, http://ehcache.org.

[41] The PostgreSQL Global Development Group. PostgreSQL, https://www.postgresql.org/.

[42] TPC Corp. TPC-C Benchmark, http://www.tpc.org/tpcc/.

[43] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–270, Dec. 2002.

A Proof of Read-after-write Consistency

Below, we provide a proof of read-after-write consistency for the write-back policy. We assume (a) a developer implements a write session correctly, transitioning the data store from one consistent state to another and (b) the data store provides strong consistency.

Definition A.1. *A read R_i may be served by a cache entry represented as a key-value pair (k_i, v_i) where k_i identifies the entry and v_i is the value of the entry. Both k_i and v_i are application specific and authored by a developer.*

Definition A.2. *A write W_j that impacts the value v_i of key k_i updates v_i to reflect the changes of W_j.*

Definition A.3. *A buffered write for a document D_i is represented as a key-value pair (k_i^{bw}, v_i^{bw}) where k_i^{bw} identifies a buffered write and v_i^{bw} stores either the final value of D_i or the pending changes to D_i. With the latter, the sequence of changes in v_i^{bw} represents the serial order of writes to D_i. This serial order is dictated either by the arrival time of writes or eXclusive (X) leases of the cache manager.*

Definition A.4. *A* mapping *inputs the key k_i for a cache entry to compute the keys $\{k_i^{bw}\}$ of the buffered writes. Without loss of generality, we assume the function f is $f(k_i) \rightarrow \{k_i^{bw}\}$.*

Theorem 1. *The write-back policy provides read-after-write consistency while applying pending buffered writes to the data store.*

We break down the theorem into the following three lemmas:

Lemma 1.1. *A read session R_i observes the values produced by the last write session that committed prior to its start.*

Proof. If v_i exists, based on Definition A.2, those writes impacting k_i that commit prior to the start of R_i updated v_i to reflect their value. R_i observes a cache hit and consumes v_i, reflecting the value produced by the latest write session. Should R_i observe a cache miss, it looks up $\{k_i^{bw}\}$ from the mapping using k_i. For each k_i^{bw}, it applies v_i^{bw} to the data store. R_i Queries the data store to compute the missing v_i and puts the resulting k_i-v_i in the cache. v_i includes changes of all write sessions that commit prior to R_i. While R_i applies v_i^{bw}, all concurrent writes that impact v_i try to acquire an X lease on k_i^{bw}, abort, and try again until R_i releases its X lease on k_i^{bw}. These writes are serialized after R_i. ■

Lemma 1.2. *Concurrent read and write sessions that reference the same buffered write are serialized.*

Proof. R_i and W_j read and write the value of the same buffered write k_i^{bw} concurrently. Two serial schedules are possible: R_i followed by W_j or W_j followed by R_i. R_i must acquire an S lease on k_i^{bw} and W_j must acquire an X lease on k_i^{bw}. S and X leases are not compatible. R_i races with W_j. If W_j wins and is granted the X lease then R_i's request for an S lease aborts and retries. If R_i wins then W_j's X lease voids R_i's S lease, forcing R_i to abort and retry. This produces the serial schedule W_j followed by R_i. In order for R_i to be serialized prior to W_j, it must commit prior to W_j requesting its X lease.

■

Lemma 1.3. *Concurrent write sessions that reference the same buffered write are serialized.*

Proof. Concurrent write sessions that impact the value of the same buffered write k_i^{bw} must acquire X leases on k_i^{bw}. Only one X lease is allowed on k_i^{bw}. When one write session acquires an X lease successfully, it incorporates its changes to v_i^{bw} while the others abort and re-try. Once the lease holder commits, it releases its X lease, allowing another concurrent write to obtain the X lease and proceed to apply its write. The X lease serializes concurrent writes, providing the isolation property. ■

Theorem 2. *The write-back policy maintains consistency of the data store in the presence of arbitrary failures of threads that apply buffered writes to the data store.*

Proof. A thread that applies a buffered write to the data store may be a background thread or a read R_i that observes a cache miss. These threads acquire an X lease on the buffered write and apply its changes to the data store. Failures of these threads cause their X lease to time out, making the buffered write available again. The buffered write may either be idempotent or non-idempotent. If it is idempotent, its repeated application to the data store does not compromise data store consistency. If it is non-idempotent, it has not yet been applied to the data store. A buffered write is always replaced with its idempotent equivalent prior to being applied to the data store. Hence, consistency of the data store is preserved. ■