MuCache: A General Framework for Caching in Microservice Graphs

Haoran Zhang*, Konstantinos Kallas*, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, Vincent Liu
University of Pennsylvania

Abstract

This paper introduces *MuCache*, a framework for extending arbitrary microservice applications with inter-service caches. MuCache significantly improves the performance of microservice graphs (commonly found in large applications like Uber or Twitter) by eliminating the need for one microservice to call another when the relevant state has not changed. MuCache is enabled by a novel non-blocking cache coherence and invalidation protocol for graph topologies that minimizes critical-path overhead. For this protocol, we prove a strong correctness result: any execution observed by the cache-enabled microservice application could have been observed by the original application without caches. Our evaluation on well-known microservice benchmarks shows that MuCache reduces the median request latency by up to $2.5\times$, and increases throughput by up to 60%.

1 Introduction

Many of today's largest web services, such as Uber, Lyft, Twitter (now X), and Meta structure their applications as microservices to enjoy the benefits of developer agility, resource provisioning, maintainability, fault tolerance, and other important metrics. Fortunately, these benefits can be reaped by smaller companies and individual developers too; recent runtimes and service mesh frameworks like Dapr [3], Envoy [4], and Istio [6] have been created to help design, deploy, and manage microservices. In a microservice architecture, applications are decomposed into a call graph of services that interact with each other and with end-users through API calls. The 'root' of the microservice call graph is typically a client-facing frontend service, while the 'leaves' are databases that store service state. This call graph is dynamic in the sense that it may be different for each request.

The call graphs of today's services are complex. Services like Airbnb and Uber consist of thousands of unique microservices that support their functionality [5, 40]. Each user request will flow through a substantial subset of these microservices. A typical Twitter request, for instance, can traverse a call graph that is 6 levels deep with significant fan-out at each level [36], and requests to popular Facebook pages often involve hundreds of servers [30]. Given the depth and breadth of modern microservice call graphs and that each edge corresponds to a network request, it is important to avoid making such calls whenever possible. One way to do so is by having each microservice reuse the responses of the services it contacts if it knows that a request will produce the same answer. Caching the responses in this fashion improves both the latency of the target request (if the branch would have been on the request's critical path) and the system's throughput as a whole (by freeing resources for other requests).

Response caching is, of course, a common technique in system design that many services are already employing to great effect. For example, a recent Alibaba analysis of its storefront microservice architecture found that around 40% of its call graphs have a depth of 3 because they hit a cache; while requests that do not hit caches reach call graph sizes of more than 40 [28]. Similarly, half of Twitter's cache clusters are used to cache intermediate computation results [38], and a study of the cache clusters at Facebook [37] reported that the cache-hit ratio for a specific cluster was more than 80%.

Unfortunately, adding caching to a microservice graph is difficult, and is something that only well-resourced companies can afford to do correctly. Indeed, no existing service mesh provides support for inter-service caching, leaving mid-size and small companies to deal with caching on their own.

The complexity of caching in microservices stems from the fact that—unlike traditional cache coherence protocols where one can reason about individual read and write operations to a target object—microservice responses are a function of the input request, the service's state, *and* an unbounded number of downstream services that are recursively called during the processing of a request. The resulting web of dependencies means that developers must carefully study the interactions between all services in the system.

Consider, for example, a user's home timeline in a social network (e.g., [14, 36]). A cached timeline response can be invalidated by changes to the user's followees, the content of contained posts, the security policies of users included in the timeline, and tweaks to the ranking algorithm; these changes can stem from requests that never touch the home timeline mixer, modifications to objects from services several hops downstream, and revisions to the control flow of dependent services. Note that, depending on the nature of the change, subtrees of the call graph may be cacheable even if the final response is not, and effective caching approaches should consider that distinction.

Given the benefits of caching and the absence of automated tools that developers can use to help them with this task, what

*Equal contribution.

can developers do today? Broadly speaking, developers today add caching by either (a) creating manual or application-specific coherence protocols, which are error-prone and fail to generalize; (b) focusing on the backend-storage layer [24, 30], which ignores the significant advantages of terminating request call graphs early; or (c) giving up on consistency and implementing simple TTL-based eviction mechanisms [1, 27, 33] that can produce stale responses.

In this paper, we propose MuCache, a framework that extends existing service meshes like Dapr to automatically provide microservice applications with inter-service caches that improve performance. Users declare the read-only methods of each service's API, and then MuCache caches the results of calls to these methods to avoid re-executing them if the data has not changed. To keep caches coherent, MuCache implements a novel lazy-invalidation cache coherence protocol for dynamic graphs of services. This new protocol has two important features. First, all cache accesses are non-blocking, so requests never need to wait for other requests to finish. This greatly reduces latency during cache hits and ensures that in the event of a cache miss, the overhead of having had to access the cache is both constant and small. Second, the protocol is provably correct and provides a very strong consistency guarantee: *all executions of a cache-enabled application are equivalent to an execution of the original application without caches*. Note that optimal tuning of these caches (e.g., selecting the optimal cache size or eviction policy) is out of the scope of this paper; instead, MuCache allows the developer to use other tools to tune each cache separately without having to worry about coherence.

Our experimental evaluation shows that MuCache achieves a median request latency reduction of up to 2.5× and a 95th-percentile tail latency reduction of up to 1.8× for well-known microservice benchmarks and applications. Additionally, MuCache increases throughput by up to 60% and allows applications with MuCache to scale as well as the original implementation. We also perform worst-case tests, artificially reducing the cache hit rate to 0%, and the results indicate that MuCache's overheads are minimal.

2 Background

In this section, we briefly review microservice architectures and the *graphs* that are formed by their interconnections.

Microservices. Applications today often comprise various services, each of which handles an incoming request, performs some task, and returns a response. This *microservice* paradigm has many benefits over prior (monolithic) architectures in which all functionality exists within a single component. For example, microservices are modular, so they can be implemented in any language and with any features so long as they expose an appropriate API (e.g., REST). This also allows teams to design, develop, deploy, scale, and optimize each microservice independently. Furthermore, each microservice can manage its own datastore, ensuring *data sovereignty*, which is important for fault tolerance and when regulations place strict access control requirements on data.

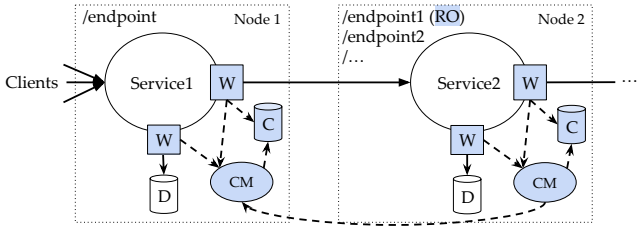

FIGURE 1—**MuCache's Architecture.** (C) denote caches, (CM) cache managers, (W) wrappers, and (D) the datastores. Wrappers are interceptor functions in the sidecar of each service. Solid arrows denote baseline communication while dashed arrows and blue components denote additions by our system. RO means read-only.

Microservice graphs. Today's microservice applications collate information from multiple backend sources and distill it into a single user interface with the help of intermediate processing functions. Such a design naturally leads to a directed graph of microservices that collectively implement the application's behavior. In this graph, the vertices are microservices, and the edges represent calls between them.

A drawback of this approach is that it incurs higher communication costs and response latency compared to monolithic systems. Whereas in a monolithic system, all invocations would typically be local function calls that leverage a machine's fast memory and native data structures, in a microservice graph, the caller needs to create the request (e.g., an HTTP request with a serialized JSON payload) and send it over the network to the callee (which may include compression and encryption), who must then deserialize it and execute it—potentially having to call further microservices. The response will incur similar overheads. For long chains of calls, the additional latency can be in the order of 100ms.

3 Goals and Overview

Given the prevalence of microservice architectures and the high costs associated with inter-service communication, we wish to design a general caching layer that avoids having one microservice call another whenever such a call is unnecessary, e.g. when the request is similar to a prior request and the state of the callee has not changed. While there is a vast design space that one could consider for achieving this high-level goal, we are grounded by a set of pragmatic requirements:

- *Correctness*: The cache should not introduce behaviors that are not part of the original application.
- *Non-blocking and low overhead*: The cache should not require blocking on the critical path, and any overhead should be minimal.
- *Dynamic graphs*: It should support microservices where the call-graph varies per-request and is not known a priori.
- *Sharding*: It should support microservices that are de-

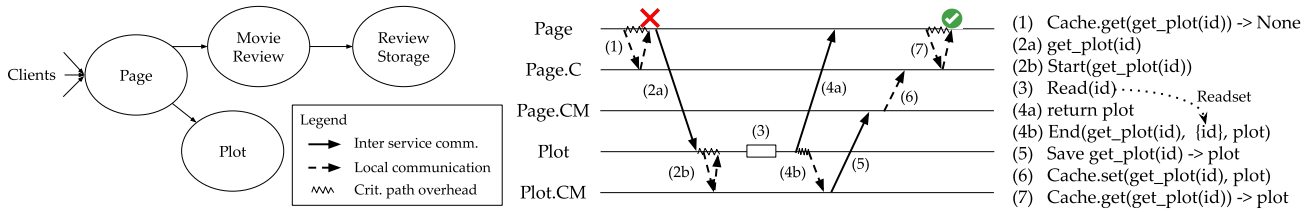

FIGURE 2—(Left) Moview Review application fragment. (Right) An example execution of this application. Each line corresponds to a different component, and arrows denote communication. (C) components are caches and (CM) cache managers.

ployed with multiple shards to enable scaleout.

- *Application and datastore agnostic*: It should not require any modifications to application logic or backend datastores for easier adoption.
- *Incremental deployment*: It should provide benefits even when only deployed on a subset of the microservice graph, e.g., if subgraphs are managed by different organizations.

3.1 Overview of MuCache

To meet the above requirements, we design and implement MuCache, a new caching framework for microservice graphs; we depict its architecture in Figure 1. In this figure, MuCache extends an application that consists of two microservices, *Service1* and *Service2*, each of which has its own datastore and exposes a set of available methods that clients or other services can call. We refer to these methods as *endpoints*, borrowing from REST terminology. Throughout the paper, when a Service1 calls a Service2, we refer to Service1 as the upstream and Service2 as the downstream.

MuCache extends each service with *wrappers* (W), a *cache manager* (CM), and a *cache* (C). The wrappers are a shim layer that intercepts all communication among services and their datastores. MuCache's wrappers are implemented on top of Dapr [3], a distributed application runtime that orchestrates service invocations and datastore accesses through its API. Dapr supports many datastores through the same API, so our wrappers inherit this compatibility without additional effort. The cache manager saves and deletes entries from the cache to maintain coherence by tracking all inter-service communication and datastore accesses through the wrappers. It is deployed as a separate executable on the same node with the service. Microservices are often sharded across multiple instances to support larger workloads; in such cases, each shard has its own cache manager and cache. MuCache does not impose any configuration restrictions on the cache which can be configured to have any eviction policy, cache size, etc. MuCache acquires knowledge of the graph topology in a decentralized way: each cache manager only knows about its immediate predecessor cache managers.

Workflow. To deploy MuCache, developers must first declare the read-only (RO) endpoints that do not mutate the datastore. If developers use REST APIs, MuCache can automatically infer this by treating 'GET' endpoints as RO. The cache will then store the responses of successful requests to the RO end-

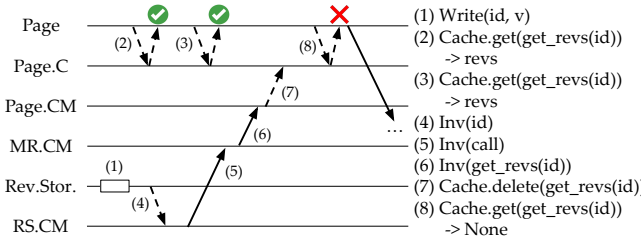

FIGURE 3—An example execution of the movie review application that includes an invalidation.

points of other services. For example, the cache of Service1 would store the return values of requests to `/endpoint1` of Service2. The cache manager of Service2 would then track all of the keys that were read during each RO call, and whenever one of these keys is modified by a write, it sends messages to all of its caller's cache managers (in this case, the CM of Service1) to invalidate the relevant cache entries.

As a concrete example, consider Figure 2 (left), which shows a fragment of a movie review application (Cf. IMDB) [24]; clients request the page of a specific movie from the `Page` service, which in turn calls the `MovieReview` and `Plot` service to compute its results. The right side of Figure 2 shows one example execution. The first time `Page` tries to get the plot for movie `id`, it does not find it in the cache (1) and then invokes `Plot`. After the call returns, the cache manager of the `Plot` service informs the cache manager of `Page` to save the return value of this call (5). For a subsequent call with the same arguments (7), the return value is found in the cache, and the `Plot` service is never contacted. Figure 3 shows an execution where some other user adds a new review to the movie `id` (1). While the write happens, different users successfully access the page of that movie from the cache (2), (3). The invalidation is propagated in the background between cache managers, until it invalidates all affected saved cache entries, including the one in the cache of `Page`.

We note that most of the processing to update and invalidate caches in MuCache is done off the critical path. With reference to Figure 2, the only operations in the critical path are steps (1), (2b), and (4b), all of which are local accesses. In particular, (1) is an access to a local cache, and (2b) and (4b) involve communication with a co-located cache manager. Furthermore, MuCache supports sharding without any communication between shards on the request processing critical path—cache managers of different shards only communi-

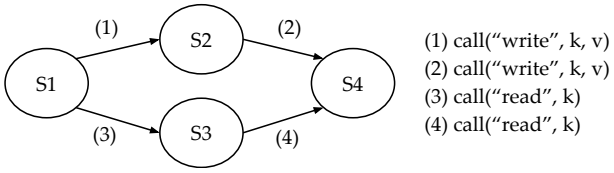

FIGURE 4—An application exhibiting a "diamond" pattern.

cate invalidation in the background. At the same time, the invalidation delays in MuCache are very small (ms)—orders of magnitude smaller than standard values of TTL used in practice to evict cache items (seconds to hours) (§7).

Correctness. The correctness condition for MuCache is based on classical refinement modulo reordering, i.e. that all behaviors exposed by a cache-enabled application are equivalent to a behavior of a cache-free version after potentially reordering independent observable events. The execution in Figure 3 is correct because it could have been observed from the original application if the write (1) had happened right after (2) and (3) since they are independent requests from different clients.

Guaranteeing correctness is challenging for call graphs with more than one path between the same two services, i.e., when a request accesses the same backend service twice in its lifetime. Figure 4 shows such an example of a 'diamond' pattern. In this example, a service S1 first calls S2, which in turn calls S4 that writes to its store. Then S1 calls S3, which calls S4 trying to read from the same value that was written by S2. It is possible that S1 could find the result of a previous call to S3 in its cache, reading a stale value, leading to an execution that would not be observable without caches. Since microservice call graphs are dynamic, this pattern cannot be identified and prevented statically (before execution). MuCache addresses this at runtime by keeping track of visited services during request processing, not checking a cached entry if it depends on a service that has already been visited.

Summary. We conclude this section by describing how MuCache satisfies the previously stated requirements:

- *Correctness*: We prove that MuCache does not introduce behaviors that are not part of the original application (§5).
- *Non-blocking and low overhead*: Cache managers do all processing in the background and the wrappers only send messages to them, never blocking for a response.
- *Dynamic graphs*: MuCache tracks dependencies to guarantee correctness in the presence of dynamic call graphs.
- *Sharding*: MuCache supports sharding without any additional communication on the critical path.
- *Application and datastore agnostic*: MuCache does not require any modification to the application or datastore code because wrappers intercept all communication.
- *Incremental deployment*: Developers can gradually declare read-only endpoints to get incremental benefits.

4 MuCache Protocol

Figure 5 shows the complete MuCache protocol for the wrappers of a single service shard and its cache manager in Python-like pseudocode. The wrapper of each service communicates with its associated cache manager through an ordered message queue (using `SendToCM`). Downstream cache managers also issue `Save`/`Inv` events to upstream ones through the same queue. Cache managers in different shards of the same service also communicate with each other when broadcasting invalidations using `SendToShardCM`.

The code on the left depicts wrapper logic run before a request starts processing (`preReqStart`), when a request has finished processing (`preReturn`), when a request reads from a key (`preRead`), before a request performs a call to another service (`preCall`), and after a request writes to a key (`postWrite`). The code on the right depicts cache manager logic, which processes events in the message queue sent by the wrappers and cache managers of other services.

Wrapper. The wrappers keep two types of state. The first is a global (per service shard) `readsets` map from request identifiers to sets of keys and call arguments, which keeps the dependencies of each pending read-only (RO) request. The second is the per-request context `ctx`, which is carried around while a request is processed. `ctx` contains (1) the id of the request (`ctx.call_id`); (2) the hash value of the request's arguments (`ctx.ca`); (3) the caller of the request (`ctx.caller`); (4) the visited services of the request and its subrequests (`ctx.visited`); and (5) whether the current request is read-only and, therefore, cacheable by its caller (`ctx.isRO`). Wrappers send a `Start(ca)` message to their associated cache manager before a request starts processing and then maintain the request readset when a read or a subrequest is performed. Once the request is complete, the entire readset, along with the call arguments, the caller, the return value, and the visited services are sent to the cache manager as an `End(ca, rs, caller, ret, vs)` message. Wrappers also send `Inv(k)` messages to cache managers after a datastore key `k` is modified. `preCall` checks the cache before invocation and returns directly upon cache hits.

Cache manager. The cache manager controls the contents of the cache. The cache manager contains two global (per service shard) state components: `saved` and `history`. The `saved` map acts as an inverted index of wrappers' `readsets` by mapping keys (or call arguments) to the corresponding service that has read (or called) them. When a key or a set of calls is invalidated, the cache manager looks up `saved` to locate all the affected upstream services and asks them to invalidate the set of relevant calls that they have cached by sending them `Inv` messages. The second state component, `history`, is a sequence of calls and invalidations used to determine whether a call can be safely cached upstream. When a request with readset `rs` is complete, the cache manager scans the history in reverse chronological order for invalidations that intersect

```
global readsets : map(Key, set(Key | CallArgs))

def preReqStart(ctx):
    if ctx.isRO:
        cid, ca = ctx.call_id, ctx.ca
        readsets[cid] = set()
        SendToCM(Start(ca))

def preReturn(ctx, ret):
    if ctx.isRO:
        cid, ca = ctx.call_id, ctx.ca
        rs = readsets.pop(cid)
        caller = ctx.caller
        vs = ctx.visited
        SendToCM(End(ca, rs, caller, ret, vs))

def postWrite(ctx, k, _v):
    SendToCM(Inv(k))

def preRead(ctx, k):
    if ctx.isRO:
        cid = ctx.call_id
        readsets[cid].insert(k)

def preCall(ctx, ca):
    if ctx.isRO:
        cid = ctx.call_id
        readsets[cid].insert(ca)
    # Check if ca refers to a read-only endpoint and if
    # the visited services are disjoint with the cache
    # item subtree
    if ca.isRO and visited_disjoint(ctx, ca):
        return cache.get(ca)
    return None
```

```
# Tracks which keys and calls will invalidate
#  which cache entries upstream
global saved : map(Key | CallArgs, map(Service, CallArgs))
# Sequence of calls and invalidations
global history : list(Call(CallArgs) | Inv(Key | CallArgs))

def startHandler(ca):
    history.append(Call(ca))

def endHandler(ca, rs, caller, ret, vs):
    # Checks if there are any invalidations
    # to the readset since the call start
    if empty([for Inv(k) in history.invs_after(Call(ca))
              if k in rs]):
        SendToCM(caller, Save(ca, ret, vs))
        saved.store(rs, ca, caller)

def invHandler(k):
    match type(k):
    case Key:
        history.append(Inv(k))
    case CallArgs:
        history.extend([Inv(ca) for ca in k])
    # Inform CMs of same-service shards
    SendToShardCMs(Inv(k)) # (see Sec. 4.1)
    # Ask all affected callers to invalidate
    affected = saved.pop(k)
    for caller, cas in affected:
        SendToCM(caller, Inv(cas))

def saveHandler(ca, ret, vs):
    save_visited(ca, vs)
    cache.set(ca, ret)
```

FIGURE 5—(Left) The wrapper code of the protocol that intercepts the start of request processing, returns, writes, reads, and calls. (Right) The cache manager code that processes work queue items sent by the wrappers and other cache managers.

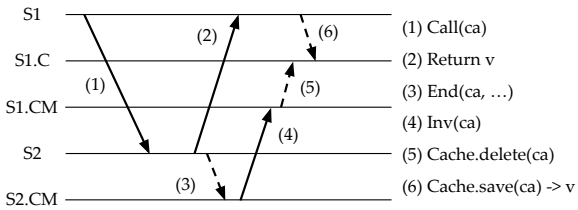

FIGURE 6—A bug that would occur if invalidate messages were allowed to overtake saves.

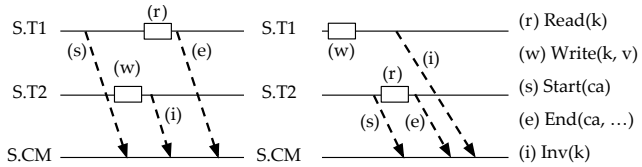

FIGURE 7—Possible imprecisions in invalidation. The three lines represent two service threads processing requests and the cache manager.

with `rs` since the call started. If there is no such invalidation, it asks the upstream cache manager to save the result. The cost of this scan is proportional to the product of request rate and average request duration, which is typically a small number. For example, a service handling 10,000 requests per second, each lasting 100 milliseconds, requires scanning several thousand items.

Saving a new cache entry. A naive method of saving a new entry involves the caller immediately saving it to the cache upon the result's arrival, rather than awaiting an explicit `Save` message from the callee's cache manager. This is not correct, as it allows the bug shown in Figure 6 where the invalidation message by the S2 cache manager "overtakes" the save done by S1, leading to the cache entry never being invalidated. Thus, it is necessary for Invs and $Save$s to not be reordered. MuCache achieves that by issuing them sequentially through the cache manager.

Invalidating an entry. Invalidations are triggered when a key used in a cached result is modified. Naively, the cache manager could track the exact order of all reads and writes to precisely track invalidations. Since requests are being processed concurrently, this would require coordination across different service threads, which would significantly slow down request processing along the critical path. MuCache relaxes the tracking of reads and writes in two ways that do not jeopardize correctness, but reduce the synchronization overhead. First, all reads of a request are gathered by the wrappers (`preRead`) and are only sent to the cache manager at the end of the request (the `rs` argument in the `End` message). To ensure correctness, the cache manager then assumes that all reads happened at the start of the call, considering the call invalid if a write happened in its duration even if it happened before the reads (Fig. 7, Left). Second, writes are intercepted in a non-atomic fashion after they have been completed (`postWrite`). This could allow for a call to start and complete in between the actual write and `postWrite`, leading to its cached response being unnecessarily invalidated (Fig. 7, Right).

Evicting an entry. There are two types of evictions in MuCache. First, a cache could fill up with entries and needs to

evict an entry to make space for new ones; in this case, the eviction is safe without any additional work since the protocol is robust to re-invalidations (i.e., it is safe to invalidate a cache entry that was previously evicted). Second, the cache manager might need to reclaim space if it is keeping track of the dependencies of many calls. It reclaims space by evicting a key or call from its `saved` dependencies and consequently sends invalidation messages to all affected calls upstream as if the key were invalidated (see `inv(k)` in Figure 5).

Garbage collection. The cache manager has two state components that grow during execution: (1) the history and (2) the dependencies. It keeps the history bounded by removing completed calls when processing an `End` request, adding minimal overhead. The protocol preserves correctness in the presence of multiple pending calls with the same arguments by removing the latest occurrence of a call start (potentially overapproximating the duration of the other calls). When the cache manager reaches a memory limit, it deletes some of its saved dependencies as long as it informs the upstream caches to evict relevant entries (similarly to a normal invalidation). The current implementation evicts dependencies following an LRU policy, though other choices could be used.

Sharding. MuCache supports sharded service deployments by attaching a cache manager to each shard; the only requirement being that read-only calls with the same arguments are always processed by the same shard (e.g., by load balancing these calls based on a hash of the call arguments). This guarantees that a single cache manager is the sole authority for the invalidations of each read-only call, ensuring that they will be the only ones to send cache-save and cache-invalidate messages for that call. The only change in the protocol is that a cache manager processing an invalidate due to a write needs to broadcast it to all cache managers of the other shards of the same service, so that they can invalidate their relevant calls (see L.25 in Figure 5). It is important to note that broadcasts only happen upon users' writes; transitive invalidations propagated upstream do not trigger broadcasts. Broadcasting out of the critical path is safe because, similarly to the single-shard protocol, overapproximating the write duration might lead to additional invalidations but not fewer. MuCache, therefore, does not add any latency overhead on the request processing critical path to support sharded services.

Handling Dynamic Call-graphs. Microservice applications can exhibit a diamond pattern (Figure 4) where a request performs multiple subrequests to the same service through its lifetime. In such applications naive caching could lead to executions that cannot be observed without caches. MuCache addresses this by keeping track of the visited services in two locations. First, each request keeps visited services in its context (`ctx.visited`); whenever a subrequest `ca` returns, the parent request adds all the visited services of the subrequest (`ca.visited`) to its own visited services (`ctx.visited`). Second, when saving a cache entry for call `ca`, the cache manager also stores the services, S', that were visited during the processing of `ca`. Before checking the cache, the wrapper checks if the downstream service has ever visited a service in S' that has also been visited by the current request `visited_disjoint(ctx, ca)`; if so, it does not retrieve the return value from the cache to preserve correctness. MuCache tracks visited services using a binary encoding that keeps its size small—less than 1 KB for 1000 services.

5 Protocol Correctness

To demonstrate the correctness of MuCache, we show that clients cannot differentiate a MuCache-enabled application from the original without caches. We give semantics to microservice applications (with and without caches) using observable execution events and traces. Events are indivisible actions (steps) that can be performed by a microservice application; examples of events include reading from a key in the datastore and receiving a response from a completed subrequest. An application can be uniquely described by the set of traces (event sequences) that can be observed in it. Two traces are said to be *equivalent modulo reordering* when all events in one trace exist in the other trace but potentially in a different order. Reorderings are necessary for our correctness theorem to allow reads and writes to proceed concurrently (as in Figure 3). In this section, we informally describe three assumptions that are central to our formal development, the first two hold for all microservice applications, and the last one is a requirement of MuCache. We then state our main theorem and give the high-level intuition for the proof. The complete formal development and proof can be found in Appendix A, which is available in the supplementary material.

(A1) Always enabled requests. Requests in a microservice application only block when waiting for subrequests that they have invoked to finish executing and there is no blocking communication across independent requests. In other words, if a trace can be observed in an application, then we can pick and execute *any* pending request, or any of its subrequests, until it produces an execution event, and the new trace will also be part of the application's set of traces.

(A2) Reordering independent events. Two events are dependent when the first event affects the execution of the second: some examples include two events that are part of the same request, or a write and a read event to the same key in a service datastore. The complete definition of dependent events is given in Appendix A. We assume that due to multithreading, independent events commute; that is, reordering any two consecutive independent events in an application trace results in a trace that can also be observed by the application.

(A3) Linearizable datastores. We assume that the datastore of each service is *linearizable* [26]: operations on an object take place atomically, in an order consistent with the operations' real-time order. For instance, if a write completes before a read begins, then the read must observe the effects

of the write and complete after it. This is necessary due to the requirement that MuCache does not modify the underlying datastore and can only observe writes to the datastore before or after they are completed. If we were to use a non-linearizable datastore a write could take effect after it returns, making it impossible to track which calls it invalidates.

Theorem 1 (Protocol Correctness)**.** *For all traces in a cache-enabled application, there exists a trace in the original application without caches, such that all the client events in the two traces are equivalent modulo reordering.*

Proof intuition. To show the theorem, we prove a stronger lemma, namely that for all cache-enabled traces, we can construct an original trace where (1) all request subtraces are the same (modulo the missing requests due to cache hits), and (2) that the application state is the same at the end of both traces. The proof proceeds by induction on the length of traces and has three phases: (1) given a trace in the cache-enabled application that ends with a cache-hit, it uses assumption (A2) to move writes that happened before the cache-hit but would later invalidate its entry to the end of the trace (together with their dependencies); (2) it then uses the inductive hypothesis to construct a trace in the original application for the prefix up to the cache-hit; and (3) it uses assumption (A1) to fill in all subrequest events that are missing due to the cache-hit, and then it fills in the writes and all their dependencies (A3), ending up with a trace that satisfies the requirement.

6 Implementation

The MuCache implementation comprises roughly 2k LoC of Go [12], including the wrappers that intercept invocations and state accesses, and the cache manager that makes invalidation and saving decisions. Communication between wrappers and the cache manager happens with ZeroMQ [16] and between cache managers with HTTP. Our current implementation uses Redis [9] as the cache, but any in-memory store could be used in its place. We use 32-bit FNV-1a [11] algorithm to compute the hash values of call arguments.

Batching. Cache managers instruct their upstream counterparts to save or invalidate cache entries by sending HTTP requests that might become a bottleneck when the load is high. To increase throughput at high loads without affecting correctness, MuCache allows batching requests that are sent upstream. At low loads, batching increases the time it takes for an invalidation to propagate through the system based on the batching timeout, which is currently set to 1ms. Batching also enables the simplification of upstream requests by canceling out operations at the sender, i.e., invalidates and saves override previous invalidates and saves on the same key. This reduces the size of requests and the number of operations upstream cache managers have to process, while incurring minimal cost since it requires a single pass over the batch.

General support. MuCache is designed to not be limited to a single communication protocol, cache, or datastore. Our wrappers are built on top of Dapr [3], a service mesh extended to also support state accesses through its API. Dapr supports custom middlewares that can be used to intercept invocations and state accesses. It also provides a common abstraction for many service communication protocols and different storage backends, allowing us to implement our wrappers once and inherit support for all the alternatives.

Dependencies between client requests. MuCache's caching protocol treats client requests as independent and allows them to be reordered, processing reads and writes from different clients without synchronization. However, this might not always be desirable, e.g., when a client request expects to see the effects of a previous request. To support this, we extend MuCache's dependencies (Sec. 4) to client requests. Specifically, when a client request is complete, visited services are included in the result, and passed to the subsequent request of the same client (if one is performed), allowing MuCache to avoid violating dependencies across client requests.

Supporting third-party services. Microservice applications often perform requests to third-party services that might not be extensible with MuCache, e.g., if they are owned by a different organization. To support such applications, MuCache allows declaring requests to third-party services as read-only using a TTL, saving their values to the cache on return, but invalidating them when the TTL has passed instead of waiting for a downstream cache manager. This setup provides caching benefits with at least as strong guarantees as if all the caches in the application were configured with a TTL, however for the complete subtrees of the microservice graph that are MuCache-enabled the guarantees are stronger.

7 Evaluation

Our evaluation aims to answer these high-level questions:

- **(Q1) Throughput and latency benefits:** Does MuCache provide throughput and latency benefits compared to other caching alternatives? Does it scale with sharding? How do cache sizes affect its performance? How are its benefits affected by the application call-graph? (§7.3)
- **(Q2) Costs:** What are the costs of deploying MuCache? What is its CPU and memory usage, total network costs, and its latency overhead on the critical path? Does the cache manager throughput become a bottleneck? (§7.4)
- **(Q3) Invalidation:** How fast can MuCache invalidate cache entries? (§7.5)

Before we answer these questions, we describe the experimental setup (§7.1) and our methodology and baselines (§7.2).

7.1 Experimental Setup

We deploy a Kubernetes [7] cluster on CloudLab [2] m510 machines that have 8-core 2.0 GHz CPUs, 64GB RAM, 256GB NVMe SSDs, and 10GB NICs. Machines run Ubuntu 20.04. The average round-trip time between servers is 0.15ms. Ex-

cept for sharding experiments, we utilize a single Kubernetes cluster where the number of worker nodes is equal to the number of services, plus one node acting as a control plane. Each service is deployed via Dapr [3] and is affinitized to a single node. We use Redis [9] configured with an LRU eviction policy as the cache. Unless otherwise noted, MuCache is configured with a sending batch size of 20 and a 1 ms timeout. Cache manager dependencies are stored in a LRU cache, with the maximum number of entries being proportional to the user cache size. In our experiments, the cache manager stores 100 dependencies per 1 MB of user cache (e.g., a user cache of 20 MB allows the storage of 2,000 dependencies).

7.2 Applications, Method, and Baselines

Throughout our evaluation we perform experiments on four open-source microservice applications, as detailed in Figure 8, along with four synthetic ones. Workloads are adapted from the original testbeds, including the dataset and request distribution. Cache sizes are set relative to the application working data set; small enough that they do not fit the entire working data set but big enough so that there is a non-negligible amount of cache-hits.

SocialMedia. A social network application (Cf. Twitter or Facebook) that provides three main endpoints, viewing a user's homepage timeline (RO), viewing a user's personal timeline (RO), and composing a post. The workload ratio is 60% homepage, 30% user timeline, and 10% compose post. The cache size for each service is set to 20 MB. When there are no new posts and each timeline contains 10 posts, the total cacheable posts are around 20 MB.

MovieReview. A movie review application (Cf. IMDB or Rotten Tomatoes) that offers two main endpoints: viewing the page of a movie (RO) and creating a review. The workload ratio is 90% viewing a page and 10% creating reviews. The cache size for each service is set to 70 MB.

HotelRes. A hotel reservation application (Cf. Booking or Airbnb) that offers two main endpoints: searching for hotels in a specific area (RO) and making a reservation. The workload ratio is 80% searching for hotels and 20% making a reservation. The cache size for each service is set to 20 MB.

OnlineBoutique. An online store application (Cf. Amazon or Walmart) that offers multiple endpoints, retrieving the store homepage (RO), updating the currency rate, viewing a product (RO), adding a product to the cart, and checking out. The workload ratio is 75% read-only (homepage, viewing products, and carts) and 25% non-read-only (updating the currency, updating the cart, checking out). The cache size for each service is set to 80 MB.

Synthetic Benchmarks. Figure 9 shows four synthetic applications: ProxyApp, a two-service app where a stateless frontend forwards requests to the backend, which in turn reads/writes to a key-value store; and three applications that extend ProxyApp with archetype call-graph patterns—chain, fan-out, and fan-in. ChainApp has four stateless services and a stateful backend. FanoutApp has a single frontend forwarding requests to four backends. FaninApp has four separate frontends, each forwarding requests to one backend.

	Benchmark	Services	LoC	RO/NonRO	Sources
1	SocialMedia	6	532	90/10	[10, 24, 32]
2	MovieReview	12	913	90/10	[13, 24]
3	HotelRes	6	608	80/20	[24]
4	OnlineBoutique	9	1,088	75/25	[8]

FIGURE 8— Real-world applications used in our evaluation.

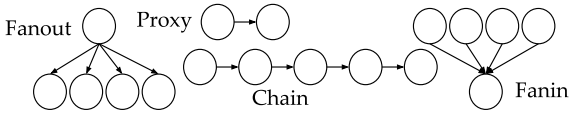

FIGURE 9—Shapes of synthetic benchmark call-graphs.

Method. We measure throughput and latency (median and 95th percentile) using the wrk2 [15] HTTP benchmarking tool. Experiments include a 30-second cache pre-warming period, followed by a 60-second testing period. Each experiment is run three times, and the average is reported. We run MuCache and the baselines with the same CPU resources; that is, MuCache's cache managers are not given extra cores but share resources with the application.

Baselines. We compare MuCache to the following baselines.

BC (Backend Cache): A baseline that lacks inter-service caching and only caches data from the backend datastore.

TTL: A baseline that reflects the current best practices for automated inter-service caching [1, 27, 33]. Caching occurs at both the backend and intermediate services. Upon invocation, the caller saves the result in the cache asynchronously without communicating with any cache manager. The caches can then evict an entry when they become full or, in the case of an inter-service cache, when a configured time-to-live (TTL) timer has expired. Cached data can be *inconsistent* and *arbitrarily stale* (depending on the TTL and access pattern).

TTL-∞: A special case of TTL that serves as an upper bound on the performance achievable by TTL implementations; cache entries never expire and are only evicted when the cache reaches maximum capacity.

7.3 (Q1) Throughput and Latency Benefits

We first measure the throughput and latency of a set of real-world applications with and without MuCache (§7.3.1). We then compare it against different TTL baselines (§7.3.2), we evaluate whether it limits throughput scalability in the presence of sharding (§7.3.3), and we evaluate whether configuring caches with different sizes and whether different application call-graphs affect MuCache's benefits (§7.3.4–7.3.5).

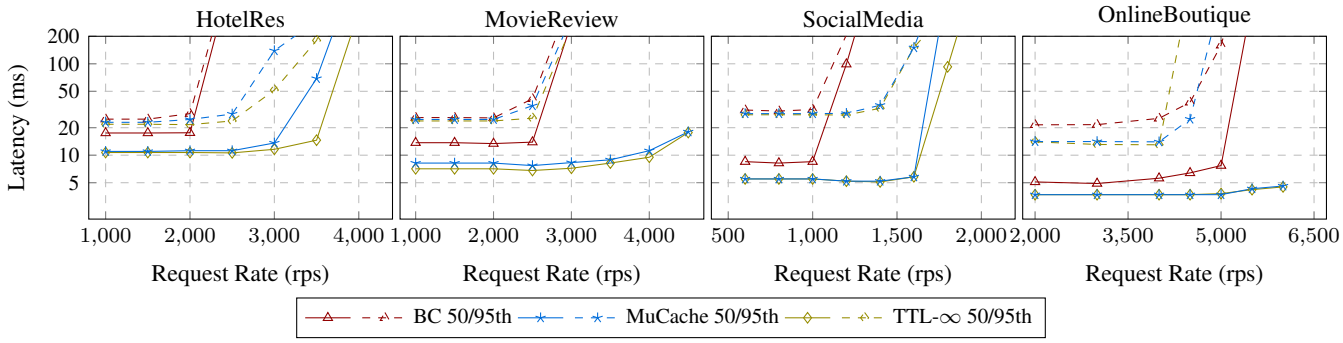

FIGURE 10—Latency and throughput of real-world applications (described in Figure 8).

7.3.1 Real-world applications

We evaluate MuCache's benefits on throughput and latency on the four open-source microservice applications. We compare MuCache against (1) BC to evaluate performance benefits over not having inter-service caches, and (2) TTL-∞ to evaluate how close MuCache is to an implementation that caches results but provides no consistency guarantees.

Results. Figure 10 shows the results, where the X-axis is request rate, and the Y-axis shows latency in ms. MuCache reduces median latency by up to 1.8× in HotelRes, 2.5× in MovieReview, 1.5× in SocialMedia, and 2.1× in OnlineBoutique. The tail latency between MuCache and BC is similar, except for OnlineBoutique, where MuCache reduces tail latency by up to 1.8× by avoiding many invocations from the Checkout service, such as retrieving product information, getting shipping quotes, etc. Furthermore, MuCache improves throughput by 1.6× in HotelRes, 1.5× in MovieReview, and 1.4× in SocialMedia, while achieving similar throughput in OnlineBoutique. Compared to TTL-∞, MuCache's median latencies are up to 1.2× higher before saturation, and MuCache's throughput is around 0.95×.

Take away. MuCache outperforms BC in terms of median and tail latency, and throughput across all workloads. MuCache also performs close to the upper bound TTL-∞. Improvements in median latency can be attributed to cache hits, while improvements in throughput are due to lower utilization of backend services.

7.3.2 Comparison with TTL baselines

Tuning TTL values for caches in real systems is complex and depends on the application requirements; suggested values could range from seconds to hours [18, 23]. To simulate that in a shorter experiment, we vary TTL from 100 ms to 10 s—values under 100 ms lead to negligible cache hits, and a TTL of 10 s is already a large fraction of the total experiment (60 s).

Results. Figure 11 shows the results. As the TTL increases from 0.1 to 10 s, median latency drops from 18.2 ms to 10.9 ms, tail latency drops from 29.3 ms to 10.9 ms, and throughput increases from 2,489 to 3,470 rps. MuCache outperforms TTL-1s (1.3× lower median latency), but is outperformed by TTL-10s (which performs similarly to TTL-∞).

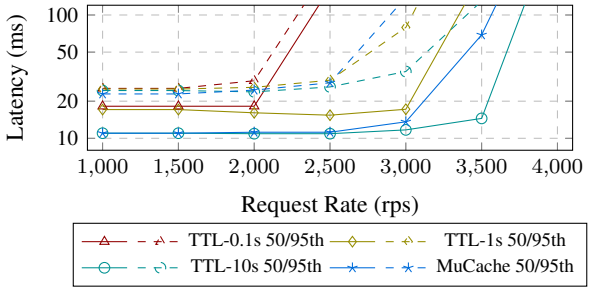

FIGURE 11—HotelRes: Latency and throughput of MuCache compared with various TTL.

Take away. Getting comparable performance to MuCache with a TTL-based caching approach requires setting the TTL to a high value (>1 s)—orders of magnitudes higher than the MuCache invalidation times (on the order of ms per call-graph depth as shown in Section 7.4.3). Furthermore, finding an appropriate TTL value is challenging for developers, as this value has implications for the correctness of the application. In contrast, MuCache requires no tuning of expiration times, and invalidations happen automatically and correctly.

7.3.3 Sharding Scalability

We evaluate the scalability of MuCache by deploying SocialMedia to multiple shards. We provision a fixed pool of machines and restrict each shard to a fixed CPU usage of 2 cores (1 running the service, 1 running the Dapr sidecar) to have multiple shards on a single machine. Each shard is deployed with its own cache manager. We compare against BC to determine whether MuCache limits scalability.

Results. Figure 12 shows the maximum throughput of the SocialMedia when deployed using 1, 2, and 4 shards, with and without MuCache. MuCache scales as well as BC (achieving 1.44×, 1.38×, and 1.37× the throughput of BC).

Take away. MuCache does not limit scalability for sharded applications as the only cost occurs in the background; when the cache manager of a shard broadcasts received writes to all cache managers that belong to the same service shards.

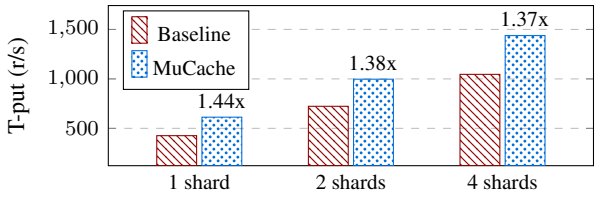

FIGURE 12—Throughput of MuCache and BC when sharding the services in SocialMedia.

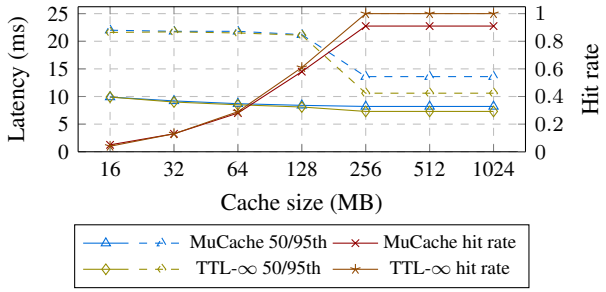

FIGURE 13—HotelRes: Impact of different cache sizes on latency (left Y-axis) and combined cache hit rate (right Y-axis).

7.3.4 Cache size effect

To evaluate how MuCache responds to the cache size of each service, we measure latency and cache hits on HotelRes with a fixed load of 1K req/s while varying the cache size from 16 MB to 1024 MB. TTL-∞ acts as an upper-bound baseline.

Results. Figure 13 shows the results. Increasing the cache size lowers the median latency of MuCache from 9.9 ms to 8.2 ms and tail latency from 22 ms to 13.6 ms; it also increases the cache hit rate from 5% to 91%. Similarly, in TTL-∞, the median latency decreases from 9.9 ms to 7.3 ms, tail latency from 21.6 ms to 10.6 ms, and cache hit rate from 5% to 100%.

Take away. Caching with MuCache reduces mean and tail latency. Furthermore, the reductions achieved by MuCache are close to those achieved by TTL-∞ across all cache sizes.

7.3.5 Application call-graph effect on performance

To evaluate how the application call-graph pattern affects the benefits of MuCache, we use the three synthetic applications in Figure 9. We use a synthetic workload with 50% cache hit rates and compare against BC.

Results. Figure 14 shows the results. For ChainApp, MuCache's median latency is 2.6–3.1× lower than that of BC, while its tail is comparable before reaching saturation. Its maximum throughput is 1.5× higher. For FanoutApp, the median latency and maximum throughput of MuCache are similar to that of the BC, but its tail latency is up to 1.6× lower. In FaninApp, MuCache improves median latency by 1.1–1.3× and 95th percentile latency by up to 1.9×; maximum throughput is 1.75× higher than BC.

Take away. MuCache provides different benefits depending on the call-graph shape. For long call-chains MuCache reduces latency by avoiding network hops; for fan-out it slightly

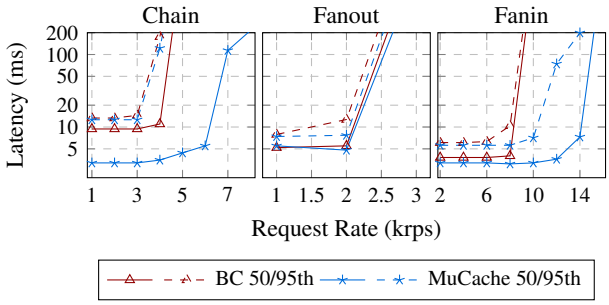

FIGURE 14—Latency and throughput of the graph shape microbenchmarks (Fig. 9).

	Benchmark	Average (MB)	Max (MB)	Cache Size (MB)
1	HotelRes	0.08	0.27	20
2	MovieReview	0.07	0.31	70
3	SocialMedia	0.02	0.09	20
4	OnlineBoutique	0.1	0.45	80

FIGURE 15—Cache manager state and cache size for each service.

improves tail-latency but not median latency since the frontend has to wait for the slowest path to respond; and when the backend is the bottleneck it improves throughput by reducing the number of requests that reach the backend.

7.4 (Q2) MuCache costs and overheads

In order to evaluate the costs of MuCache, we measure its CPU, memory, and network usage (§7.4.1), its latency overhead on the critical path (§7.4.2), and the cache manager's throughput and whether it can be a bottleneck (§7.4.3).

7.4.1 Memory / CPU / Network costs

We evaluate MuCache's memory cost on all four applications and its CPU and network usage on HotelRes. We evaluate MuCache's network usage by measuring data transfer between nodes using `iftop`. We measure the memory cost of each cache manager instance as the average size of its state (history and dependencies) and CPU cost as the average CPU usage of each service during the experiment. We use standard cache sizes and load (2K req/s for HotelRes, 2.5K req/s for MovieReview, 1K req/s for SocialMedia, and 3.5K req/s for OnlineBoutique) for 300 seconds.

Results. Figure 15 shows the cache manager state size and the cache size across services. The average size of the CM state across services ranges from 0.1–0.4% of the cache size per service. Garbage collection plays an important role in keeping the memory usage low: without GC, the CM state in HotelRes goes up to 5 MB in 1 minute. Figure 16 shows the average CPU usage of each service during the experiment. Usage is broken down between the service logic, the Dapr sidecar, and the cache manager. The average CPU usage across services with and without MuCache is 4.2 and 5.1 cores respectively. The average CM CPU usage across services is 0.5 cores. The

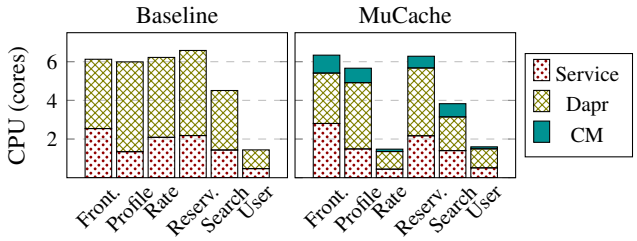

FIGURE 16—CPU usage per service for HotelRes.

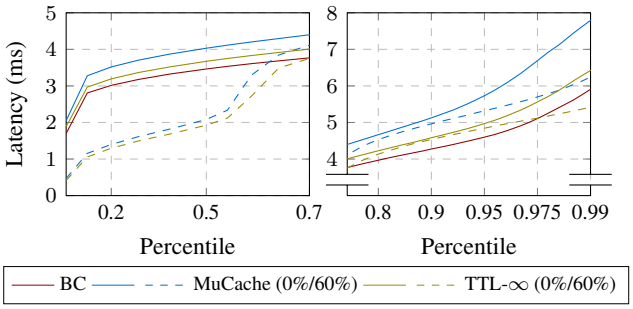

FIGURE 17—Latency distribution w.r.t. hit-rate for ProxyApp. Solid and dashed lines show the latencies when the hit rate is 0% and 60% respectively. Split in 70th percentile for clarity.

average network usage per service without MuCache is 9.0 MB/s, while the average with MuCache is 6.6 MB/s, of which cache managers use 2.9 MB/s.

Take away. Memory costs are low compared to the cache size (<0.4% on average). The CPU usage of MuCache is 13% of the total service CPU on average while at the same time reducing the total CPU usage of the whole application due to some backend services being less utilized because of cache hits in the frontend. Though cache managers use some bandwidth to save/invalidate caches, MuCache reduces the total network usage by 27% due to local cache hits.

7.4.2 MuCache latency overhead

We evaluate MuCache's latency overhead by focusing on ProxyApp, which performs minimal work, to measure the worst-case overhead. We create a synthetic workload with 0% and 60% cache hit rates and compare against (1) BC to evaluate the overhead over no caches when there are no hits, and (2) TTL-∞ to evaluate the wrapper overhead.

Results. Figure 17 shows the complete request latency distribution. We report overheads as absolute values because they are constant and independent of the work that the services do. For a hit rate of 0%, MuCache's median latency (4 ms) is 0.5 ms higher than BC and 0.3 ms higher than TTL-∞, while the 95th-percentile (5.7 ms) is 0.9 ms and 0.5 ms higher respectively. When the hit rate is 60%, MuCache's median and 95-th percentile latencies are 0.15 ms and 0.5 ms higher than TTL-∞. When the hit rate is 60%, MuCache median latency is 1.4 ms better than BC (3.5 ms to 2.1 ms).

Take away. Even in a worst-case scenario (an application that performs minimal work), MuCache imposes a low ($\sim$10%) latency penalty on cache misses.

Batch Size	1	2	5	10	20	50
Throughput (krps)	19.2	26.5	44.5	74.7	75.2	74.6

FIGURE 18—Batching effects on cache manager throughput.

Chain Size	2	3	4	5
Mean invalidation time (ms)	3.94	6.13	8.41	10.66

FIGURE 19—Invalidation time for different chain sizes.

7.4.3 MuCache's throughput

To determine whether MuCache's cache manager can be a bottleneck, we measure the its maximum throughput on the ProxyApp and load the backend's cache manager directly because the backend service becomes the bottleneck otherwise. The load is 80% read-only requests and we vary the batch size of the HTTP sending buffer between cache managers.

Results. Figure 18 shows the throughput in terms of the number of events the cache manager processes per second. Without batching, the cache manager has a throughput of $\sim$19K events per second, while gradually increasing the batch size up to 20 improves it to $\sim$75K events per second.

Take away. The cache manager has a reasonably high throughput and is not the bottleneck even for an application with minimal computation. To further increase throughput, developers may deploy multiple shards for each service.

7.5 (Q3) Invalidation time

We evaluate the time needed for invalidations to reach the root of the call-graph, namely the frontend service, by measuring the observed inconsistency window [19], the elapsed time between the write happening in the backend and the invalidation becoming visible in the frontend. The invalidation time in our experiment is determined solely by the depth of the call graph. To measure the increase in invalidation time per hop, we conducted experiments on a microservice chain consisting of 2 to 5 services, which represents the typical depths of call-graphs in the applications that we studied.

Results. Figure 19 shows the results. For a two-service application, the invalidation time is $\sim$4 ms; for a five-service application it is $\sim$10 ms. Each additional service in the chain increases invalidation time by $\sim$2.2 ms.

Take away. MuCache's invalidation time is $\sim$2.2 ms per call-graph hop—orders of magnitude smaller than the typical invalidation times observed in TTL-based approaches (which range from seconds to hours [18, 23]).

8 Related Work

Caching in microservice applications. Several works study cache usage in real-world microservices, including work from Alibaba [28], Twitter [38], and Facebook [37]. These papers

confirm that caches are heavily used in microservice applications and provide significant performance benefits, but only mention manual, ad-hoc, or inconsistent coherence schemes and do not propose an automatic way to manage these caches.

Caching frameworks for web services. There is a lot of work on caching frameworks for web services for both static and dynamic data. These frameworks focus on three key aspects: (1) content admission, (2) cache size management, and (3) invalidation and data freshness (for a more detailed classification see a recent survey [29]). The first two aspects are orthogonal to our work since we do not focus on optimizing the performance of a cache given a specific workload, but rather propose a general system for keeping caches coherent in a microservice setting. To the best of our knowledge, all frameworks that focus on invalidation (e.g., [21, 22, 31]) are designed as a single cache layer on top of a database without taking into account the inter-service caching.

Cache coherence protocols. There is extensive literature on cache coherence protocols (see survey [34]), none of which considers inter-service caching. Lazy caching [17] exploits the fact that writes do not always require exclusivity (M or E in MOESI [35]), allowing cores to perform concurrent buffered writes, albeit blocking reads to ensure that dependencies are not violated. Our work extends this insight by avoiding all blocking communication on the request's critical path—allowing writes downstream without immediately informing the upstream caches and without blocking on reads.

Incremental computation. Caches are also used to enable incremental and reactive computation: some examples include Reactive Caching [20], Noria [25], and Diamond [39]. Reactive Caching proposes caches for graphs of single-threaded services to support reactive computation, i.e., writes downstream are propagated upstream to refresh the results. Noria is an incremental stream processing engine that uses caches for fast propagation of updates in a dataflow. Both differ from our work in two ways: (1) they only provide eventual consistency that violates dependencies when there are multiple paths between two services (see Fig. 4); and (2) they do not support true multi-threading, as Noria limits writes to a single thread and Reactive Caching only supports single-threaded services. Diamond is a system that automates data management for distributed reactive applications by providing reactive transactions to clients. Similarly to MuCache, Diamond reactively informs clients about data invalidations in the backend store, but in contrast to MuCache it does not support service graphs.

9 Discussion and Limitations

Supporting transactions and non-KV stores. Our implementation does not currently support transactions or non-KV stores. Supporting single-service transactions would require that the wrappers perform the `postWrite` after the transaction has completed to overapproximate the time that the write operation completed. Supporting multi-service transactions would be more challenging since caches should not violate transactional guarantees, which would require additional synchronization in the protocol. Supporting non-KV stores, such as relational databases, would require monitoring the dependencies of read-only calls and determining when to invalidate cache entries, which could be done by leveraging the expressive semantics of SQL (as in the case of Noria [25]).

Supporting weaker consistency datastores. The correctness of MuCache depends on the datastores being linearizable; MuCache needs to be sure that after a write has completed, it has taken effect in the database. Being able to determine the order of reads and writes by intercepting the datastore accesses is necessary so that MuCache is database-agnostic (see requirements in Section 3). Supporting weaker consistency datastores would likely require a more intrusive design with modifications to a datastore—tightly integrating wrappers in the store to provide additional metadata to the cache managers about the precise order of reads and writes—forfeiting the generality of being database-agnostic.

Application debuggability. Extending an application with MuCache provides performance benefits and does not affect the application behavior but adds complexity to the end-to-end deployment and therefore increases the effort required to maintain and debug it. This is an inherent software engineering challenge—the bigger a codebase is, the harder it is to maintain it. A direction for future work that could help address this is to integrate MuCache with existing distributed tracing and debugging tools for microservices, so that engineers have visibility on MuCache's state and actions.

Write-intensive workloads. Even though a service might offer a read-only endpoint, its workload might be write-intensive, leading to overheads without the accompanied benefits if extended with MuCache. Developers can currently manually detect such cases and avoid declaring those endpoints as read-only, but it would be interesting to explore whether MuCache can be extended with an adaptive monitoring mechanism that only enables caching if the read-write ratio of a service is above some threshold.

Sharding. MuCache requires hard affinity sharding of read requests to ensure correctness, i.e., all read-only calls with the same arguments need to be processed by the same shard. Write requests have no such limitation and can be dispatched to any shard. An interesting avenue for future research would be to lift the requirement for hard affinity, allowing for more flexible load balancing and autoscaling.

Acknowledgments

We thank the NSDI 24 and SOSP 23 reviewers, our shepherd, Gábor Rétvári, as well as Achilles Benetopoulos, Akis Giannoukos, Jiali Xing, Nathaniel Hoaglund, and Nikos Vasilakis, for discussions and feedback on the paper. This work was partially supported by NSF awards CCF 2124184, CNS 2107147, and CNS 2321726.

References

[1] Caching Guidance - Azure Architecture Center. https://learn.microsoft.com/en-us/azure/architecture/best-practices/caching.

[2] CloudLab - A testbed for cloud computing research. https://www.cloudlab.us/.

[3] Dapr - Distributed Application Runtime. https://dapr.io/.

[4] Envoy Proxy. https://www.envoyproxy.io/.

[5] From Monolith to Microservices: How to Scale Your Architecture. https://www.youtube.com/watch?v=N1BWMW9NEQc.

[6] Istio Service Mesh. https://istio.io/latest/about/service-mesh/.

[7] Kubernetes - An open-source container orchestration system. https://kubernetes.io/.

[8] Online Boutique – Microservices Demo. https://github.com/GoogleCloudPlatform/microservices-demo.

[9] Redis - An open-source in-memory data store. https://redis.io/.

[10] Rutgers Social Network Graph. https://networkrepository.com/socfb-Rutgers89.php.

[11] The FNV Non-Cryptographic Hash Algorithm. https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-17.html.

[12] The Go programming language. https://go.dev/.

[13] The Movie Database. https://www.themoviedb.org/.

[14] Twitter's recommendation algorithm. https://github.com/twitter/the-algorithm.

[15] wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[16] ZeroMQ - An open-source universal messaging library. https://zeromq.org/.

[17] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1993.

[18] AWS. Caching Best Practices. https://aws.amazon.com/caching/best-practices/, 2023.

[19] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Workshop on Middleware for Service Oriented Computing (MW4SOC)*, 2011.

[20] Sebastian Burckhardt and Tim Coppieters. Reactive caching for composed services: polling at the speed of push. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2018.

[21] K Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of the ACM SIGMOD Conference (SIGMOD)*, 2001.

[22] Jim Challenger, Arun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 1999.

[23] Cloudflare. Edge and Browser Cache TTL. https://developers.cloudflare.com/cache/how-to/edge-browser-cache-ttl/, 2023.

[24] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[25] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M Frans Kaashoek, and Robert Tappan Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[26] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), July 1990.

[27] Joydip Kanjilal. Scaling microservices architecture using caching. https://www.developer.com/design/scaling-microservices-using-cache/, 2021.

[28] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2021.

[29] Jhonny Mertz and Ingrid Nunes. Understanding application-level caching in web applications: a comprehensive introduction and survey of state-of-the-art approaches. In *ACM Computing Surveys (CSUR)*, 2017.

[30] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[31] Dan RK Ports, Austin T Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache.

In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[32] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the AAAI conference on artificial intelligence (AAAI)*, 2015.

[33] Irfan Saleem, Pallavi Nargund, and Peter Buonora. Data caching across microservices in a serverless architecture. https://aws.amazon.com/blogs/architecture/data-caching-across-microservices-in-a-serverless-architecture/, 2008.

[34] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. In *IEEE Computer*, 1990.

[35] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *ACM SIGARCH Computer Architecture News (SIGARCH)*, 1986.

[36] Alex Xu. Twitter architecture 2022 vs. 2012. what's changed over the past 10 years?, Nov 2022.

[37] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Characterizing Facebook's Memched Workload. In *IEEE Internet Computing*, 2013.

[38] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[39] Irene Zhang, Niel Lebeck, Pedro Fonseca, Brandon Holt, Raymond Cheng, Ariadna Norberg, Arvind Krishnamurthy, and Henry M Levy. Diamond: Automating data management and storage for wide-area, reactive applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[40] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of Large-Scale microservice architectures. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2022.

A Detailed Protocol Correctness

Preliminaries. We start with some basic notation:

- n_S denotes a service name
- ca denotes the arguments of a service call, including the name of the service (which can be extracted using name(ca)) and the endpoint.
- $i \in R$ denotes request identifiers (each request has a unique i). The service name and the arguments to the request can be extracted using name(i) and ca(i). We will define a binary relation $sr \subseteq R \times R$ that determines when a request is spawned by another request. We can also define sr^* as the reflexive transitive closure of sr. There is also a client(i) predicate, which returns true for requests that are initiated by a client.
- v denotes a return value
- k denotes a key that indexes values in the state of a service
- rs(i, t) is a function that returns all of the keys that a particular request (and all of its subrequests) have read in trace t. We will often ignore t when it is obvious which trace we refer to.

Events and traces. We will describe microservice applications and their executions using traces, i.e., sequences of events that describe application actions. We are only interested in events that describe interactions between services and other services and actions on their states. We call the set of all events Σ, and we now define all events in it.

- $\text{Req}_i(ca)$ denotes the start of processing of a single request with id i and arguments ca.
- $\text{Ret}_i(v)$ denotes that a request with id i has finished processing and is returning value v.
- $\text{Read}_i(k, v)$ denotes that request with id i performed a read of key k from its state and returned v.
- $\text{Write}_i(k, v)$ denotes that request with id i performed a write with value v to key k of its state.
- $\text{Call}_i(ca, i')$ denotes that request with id i performed a call to another service with arguments ca and the request id of that internal request is i'.
- $\text{Resp}_i(v, i')$ denotes that request with id i received a response with value v from a finished call with id i'.

We represent the set of all events for a request with identifier i as Σ_i and the set of all read (or write) events as Σ_R (or Σ_W). We also define a set of output events $\Sigma_O = \{\text{Ret}_i(v) : \forall i, v\} \cup \{\text{Read}_i(k, v) : \forall i, k, v\} \cup \{\text{Write}_i(k, v) : \forall i, k, v\} \cup \{\text{Call}_i(ca, i') : \forall i, ca, i'\}$ that are events that are determined by the program when processing a single request, and input events $\Sigma_I = \{\text{Req}_i(ca) : \forall i, ca\} \cup \{\text{Resp}_i(v, i') : \forall i, v, i'\}$ that are events that are given as inputs to the processing of a single request. Finally, we can define the set of client events $\Sigma_C = \{\text{Req}_i(v) : \forall i, \text{client(i)}\} \cup \{\text{Ret}_i(v) : \forall i, \text{client(i)}\}$ We can now describe complete executions of microservice applications using traces t, i.e., sequences of the above events. We can project all events of a trace t from a particular set Σ using $t[\Sigma]$, e.g., $t[\Sigma_W]$ are all the write events in a trace. Note that this projection creates an ordered sequence of events by maintaining the trace order.

Applications and Assumptions. We can now define the behavior of a microservice application $P \in \mathbb{P}$ using its execution traces, $[\![P]\!] \subseteq \Sigma^*$, and state some assumptions on these traces. First of all, an application determines the processing of each request using the step : $\mathbb{P} \times R \times \Sigma^* \times (\Sigma_O \cup \{\bot\})$ relation, that determines the next step of the processing of a request, or $\bot$ if the request is waiting for a response or hasn't started yet. Now we define what it means for a trace to be well formed.

Property 2 (Well-formed traces)**.** *All traces $t \in [\![P]\!]$ are well-formed, i.e., for each trace t the following properties hold: (1) $\text{Req}_i(ca)$ are the first events for any request i and $\text{Ret}_i(v)$ are the last; (2) for each $i \in t$ there exists a unique $\text{Req}_i(ca)$ and at most one $\text{Ret}_i(v)$; (3) a $\text{Req}_i(ca)$ always comes after a $\text{Call}_i(ca, i')$ except in the case of client requests; (4) a $\text{Resp}_i(v, i')$ always comes after a $\text{Call}_i(ca, i')$ and $\text{Ret}_{i'}(v)$; (5) for all $\text{Call}_i(ca, i')$, $sr(i, i')$; and for all prefixes $t' = t_0.e$ with $e \in \Sigma_I$, either step(P, i, t_0, e) or step$(P, i, t_0, \bot)$; (6) for all $e \in \Sigma_C$ for any $i \in t$, s.t. client(i) holds, then $\nexists \text{Call}_{i'}(ca', i) \in \Sigma_C$.*

The last requirement relates the step relation with the traces, i.e., each event in the trace is the result of stepping a request or a request start or response. We also know that the events in a trace are equivalent up to an injective renaming of request identifiers.

Property 3. *For any microservice application P, for all traces $t \in [\![P]\!]$ and for all $i \in t$, then for any $i' \notin t$ we can construct a new trace $t' = t\,[i \mapsto i']$, s.t. $t' \in [\![P]\!]$.*

In addition to the above, we also know that requests are always enabled in microservice applications, i.e., a pending request can always take a step.

Definition 1 (Pending Requests)**.** *We say that a request $\text{Req}_i(ca)$ is pending in a trace t iff $\text{Ret}_i(v)$ does not exist in t.*

Property 4 (Request Step Always Enabled)**.** *For any microservice application P, for all traces $t \in [\![P]\!]$, and for all pending requests $\text{Req}_i(ca)$ for some i, there exists a trace $t' \in [\![P]\!]$ such that $t' = t.e_i$, where $e_i \in \Sigma_{i'}$ and $sr^*(i, i')$.*

Property 4 means that requests are always enabled to take a step, sometimes through their subrequests. This is a valid assumption for microservice applications since they are multithreaded, and therefore a single request can not block other requests from proceeding, and a request can only block while waiting for a response from its subrequests. Note that this assumption requires that the network does not drop requests, i.e., calls eventually lead to request starts and that returns eventually lead to response events.

We also know that the values of read events depend on the latest write to the same key or the original value.

Property 5 (Read return value)**.** *For all applications P, request identifiers i and traces t s.t. step$(P, i, t,$ Read$_i(k, v))$ holds, then either $\exists i'$, Write$_{i'}(k, v) =$ last$(t[\Sigma_{W(k)}])$ or $v = \bot$.*

Intuitively, this means that writes are immediately visible to reads, thus that the underlying stores are linearizable, which is a valid assumption for most key value stores.

We can now define read-only calls, that is calls that never perform writes (even in their subrequests).

Definition 2 (Read-only requests)**.** *Given an application P a request with request identifier i and call arguments ca, i.e. $ca(i) = ca$, is read-only for this application iff for all traces $t \in [\![P]\!]$, and for all i' such that $sr^*(i, i')$, it holds that $t[\Sigma_W \cap \Sigma i'] = \emptyset$. We define a predicate $RO(i)$ that holds for read-only requests.*

State. We represent the state of an application as $\sigma \in \mathbb{D}$. Concretely, a state σ is a tuple of maps from keys to values, one for each service. We define the function $S : \Sigma^* \to \mathbb{D}$ that returns the state of an application after the trace t. Due to Property 5 the state at each point in the execution depends on the prefix of write events and the starting state. We assume that all executions start from the same starting state σ_0.

Caching. Up to this point we have established all important properties of microservice applications without mentioning caches. A cache-enabled application $\widetilde{P}$ can be similarly defined by its execution traces, $[\![\widetilde{P}]\!] \subseteq \widetilde{\Sigma}^*$, where $\widetilde{\Sigma}^*$ is a superset of the set of events of applications without caches, i.e. $\Sigma \subseteq \widetilde{\Sigma}$. The additional cache related events are defined as followed:

- CacheHit$_i(ca, v)$ denotes a cache-hit that replaces a Resp$_{i'}(v, i)$ for some i' (also conforming to its well-formedness conditions Property 2).
- Save(n_S, i, v) denotes that the cache of service n_S has saved the value v for request i with call arguments ca(i).
- Inv(n_S, i, i') denotes an invalidation of the cache of service n_S with ca(i') from a write with identifier i.

Essentially, a cache-enabled application $\widetilde{P}$ is a transformation of a regular microservice application P. We know that our protocol does not affect the stepping of requests other than allowing some calls to return immediately with call hits. We can also lift the step relation to account for cache-enabled applications. The lifted step relation describes the logic of our cache coherence protocol.

Property 6 (Cache Stepping)**.** *For any application P the transformed $\widetilde{P}$ can step, i.e. step$(\widetilde{P}, i, t, e)$ holds, if*

- *step(P, i, t, e) when $e \in \Sigma$ or*
- *$e =$ Save(n_S, i', v) and $\exists$Ret$_{i'}(v) \in t$ or*
- *$e =$ Inv(n_S, i', i'') and $\exists$Write$_{i'}(k, v) \in t$ with $k \in$ rs(i'').*
- *$e =$ Inv(n_S, i', i'') and $\exists$Inv$(n_S, i', i''') \in t$ with ca$(i''') \in$ rs(i'').*
- *$e =$ CacheHit$_i(ca, v)$ and $\exists$Save$($name$(i), i', v) \in t$ and $\nexists$Inv$($name$(i), i'', i''') \in t$ and $ca = ca(i') = ca(i''')$ between the save and the cache-hit.*

Intuitively, Property 6 means that the cache-enabled application does not affect the next steps of any specific request other than sometimes finding a result in the cache.

Definition 3 (Dependency)**.** *We say that event $e' \in \Sigma_{i'}$ is a dependency of $e \in \Sigma_i$ in a trace t if e' is after e and if either:*

- *$i = i'$, i.e. the two events are part of the same request*
- *$e =$ Call$_i(ca, i')$ and $e' =$ Req$_{i'}(ca)$ $i \neq i'$ and $sr^*(i, i')$, i.e., the second event is a part of a subrequest of the first event*
- *$e =$ Ret$_i(v)$ and $e' =$ Resp$_{i'}(v, i)$, i.e., the events are a pair of return and handle response.*
- *$e =$ Write$_i(k, v)$ and $e' =$ Read$_{i'}(k, v')$ or $e =$ Write$_i(k, v)$ and $e' =$ Write$_{i'}(k, v')$ or $e =$ Read$_i(k, v)$ and $e' =$ Write$_{i'}(k, v')$, i.e., read and write events to a key k are dependencies of a prior write to the k and write events are dependencies of a prior read.*
- *$e =$ Ret$_i(v)$ and $e' =$ Save(n_S, i, v) for some n_S*
- *$e =$ Write$_i(k, v)$ and $e' =$ Inv(n_S, i, i') for some n_S*
- *$e =$ Inv(n_S, i', i''') and $e' =$ Inv(n_S, i', i'') with ca$(i''') \in$ rs(i'').*
- *$e =$ Save$($name$(i), i', v)$ and $e' =$ CacheHit$_i(ca, v)$ where ca$(i') = ca$*
- *$e =$ Save(n_S, i', v) and $e' =$ Inv(n_S, i, i'') for some i and ca$(i') =$ ca(i'')*

We will use deps(e) to refer to all the transitive dependencies of an event e. We now state a final assumption on application traces, namely that two independent events can be commuted.

Property 7 (Commute independent events)**.** *For any trace $t \in [\![P]\!]$ with $t = t_0.e.e'.t_1$ and $e' \notin deps(e)$, then $t' = t_0.e'.e.t_1$ can also be observed by the application, i.e. $t' \in [\![P]\!]$.*

This holds because in microservice applications independent requests do not affect each other except through reads and writes to the same key in the same service datastore.

We are now ready to state the main theorem that describes the correctness of our protocol.

Theorem 8 (Protocol Correctness (corresponds to Theorem 1))**.** *For all traces t in a cache-enabled application $[\![\widetilde{P}]\!]$, there exists a trace t' in the original application without caches $[\![P]\!]$, such that their respective client events are equivalent (but potentially reordered), i.e., $\forall i\ t[\Sigma_{C(i)}] = t'[\Sigma_{C(i)}]$.*

This makes sense because correctness is only relevant from the perspective of the clients and not all of the internal events that an application performs. Actually, the cache implementation does not contain the same traces because some calls return immediately on cache-hits without triggering all the internal events. In order to prove this theorem, we show that something stronger holds, a lemma that is stated below. Before stating it, we need to define what it means for an event in the cache-enabled event set to be equivalent to the original one.

Definition 4 (Equivalent events)**.** *Equivalence between a cache-enabled event e_c and an original event e (denoted with $e_c \simeq e$) is defined as followed:*

- *if $e \in \Sigma$ and $e_c \in \Sigma$ are the same event or*
- *$e_c = CacheHit_i(ca(i'), v)$ and $e = Resp_i(v, i')$*

We can lift the equivalence relation of events to account for sequence of events in a straightforward way.

Lemma 1. *Given an arbitrary trace $t \in [\![\widetilde{P}]\!]$ we can construct a trace $t' \in [\![P]\!]$ such that (i) the states at the end of the traces are the same for both traces, i.e. $S(t) = S(t')$, and (ii) for all i, $t[\Sigma_i] \simeq t'[\Sigma_i]$ modulo the missing events due to the cache-hits.*

At a high-level the proof proceeds by constructing a t' from t in the missing events and also by moving some write events later in the trace. Theorem 8 follows directly from Lemma 1 since client events will be the same in both traces.

Proof sketch. We will proceed by induction on the size of traces and for the inductive case we will focus on the only interesting scenario where the trace t ends with a cache-hit event $\text{CacheHit}_i(ca, v)$, because these are the only events for which the effects of our cache-subsystem are observed by the rest of the application. For illustrative purposes we extend traces with the state of all services σ_n between each event.

$$t = t_0|_{\sigma_n}.\text{CacheHit}_i(ca, v)$$

For this cache-hit to have happened, the step relations implies that there must exist some $\text{Save}(n_S, i', v)$ before it, such that $\text{name}(i') = n_S$. Similarly, for the cache save to have happened, there must have been a completed request with call arguments $ca = \text{ca}(i')$.

$$t = \cdots .\text{Req}_{i'}(ca)|_{\sigma_1}.\cdots .\text{Ret}_{i'}(v)|_{\sigma_2}.\cdots .\\ \cdots .\text{Save}(n_S, i', v)|_{\sigma_3}.\cdots|_{\sigma_n}.\text{CacheHit}_i(ca, v)$$

Given Property 2 (extended in a straightforward way to support cache events), we know that $t[\Sigma_{i'}]$ can be produced by the step relation. The inductive hypothesis and the fact that t is finite ensure the equivalence of the traces even in the presence of cache-hits for subrequests of the original request. We will now do a case analysis on the existence of a $\text{Write}(k, v_1)$ where $k \in \text{rs}(i')$ between $\text{Req}_{i'}(ca)$ and $\text{CacheHit}_i(ca, v)$.

No such write exists. If no such write exists, then $\sigma_1|_{\text{rs}(i')} = \sigma_2|_{\text{rs}(i')} = \ldots = \sigma_n|_{\text{rs}(i')}$. Then, we can construct a trace $t_1 \in [\![P]\!]$ using the inductive hypothesis and by replacing $\text{CacheHit}_i(ca, v)$ with $\text{Call}_i(ca, i'')$ for some fresh i'' (due to Property 6).

$$t_1 = \ldots|\sigma_n.\text{Call}_i(ca, i'')$$

Then, given that $\sigma_1|_{\text{rs}(i')} = \sigma_n|_{\text{rs}(i')}$ and that Properties 5 and 3 hold, we can construct the same request steps t_c as in the original trace ($t[\Sigma_{i'}]\,[i' \mapsto i'']$) using the step relation, ending up with a trace $t_2 \in P$ such that:

$$t_2 = t_1.t_c.\text{Resp}_i(v, i'')$$

Since $\text{CacheHit}_i(ca, v) \simeq \text{Resp}_i(v, i'')$ and read-only requests do not modify the state, we are done with this case.

Write exists. We now need to focus on the case where a write $\text{Write}(k, v_1)$ with $k \in \text{rs}(i')$ exists between $\text{Req}_{i'}(ca)$ and $\text{CacheHit}_i(ca, v)$. We can first show that the write is between $\text{Save}(n_S, i', v)$ and $\text{CacheHit}_i(ca, v)$, because if it was earlier, it would have been processed by the cache manager, prohibiting $\text{Save}(n_S, i', v)$ to have happened. However, there could be an invalidate between the cache save and the cache-hit that has originated from a previous write in another service between $\text{Req}_{i'}(ca)$ and $\text{Save}(n_S, i', v)$. We can now use Property 7 to move all writes together with their dependencies to the end of the trace to get a trace t_w.

$$t_w = \cdots .\text{Save}(n_S, i', v)|_{\sigma_3}\cdots|\sigma_n.\text{CacheHit}_i(ca, v).\cdots .t_{wd}$$

where t_{wd} contains all the writes and their dependencies. This is possible because $\text{CacheHit}_i(ca, v)$ is not a dependency of the writes between the save and the cache-hit; if it was, there must have been a subcall to the service where the write happened, which would have been caught by our dependency tracking (see Section 4). Second, all i' events are not dependencies of the writes between $\text{Req}_{i'}(ca)$ and $\text{Save}(n_S, i', v)$ because (1) i' is read-only (so it cannot have performed those writes or subcalls that performed those writes), and (2) the

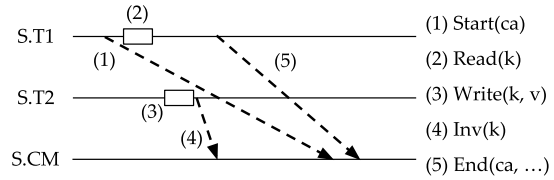

FIGURE 20—A bug that would occur if `preReqStart` does not wait until the `Start` event is added in the CM workqueue.

write happened after the call to start i'.

We can now follow the same reasoning as in the no-write-exists case. We first use the inductive hypothesis on the prefix until the cache-hit and Property 6 to get the following trace:

$$t_1 = \cdots | \sigma_n.\text{Call}_i(ca, i')$$

We then construct the original trace that caused the save using the step relation (like in the no-write-exists case) to get

$$t_2 = \cdots | \sigma_n.\text{Call}_i(ca, i'). \cdots .\text{Resp}_i(v, i')$$

Finally, given that both prefixes and states are the same for t_w and t_2, we can use Property 4 to step all the writes and their dependencies to acquire the same exactly events as the suffix of t_w, proving that the states are the same and the traces for each request in the end are equivalent.

B MuCache protocol design details

B.1 Waiting for events to be added in the queue

It is crucial that the caller waits until the event is added to the queue when sending a `Start` message, otherwise the bug shown in Figure 20 could occur. In this scenario, thread T1 of a service S starts processing a RO request `ca` before waiting for the `Start(ca)` event to be added to the workqueue. In the meantime, another thread T2 of S performs a write which invalidates the results of the call `ca`. However, since the `Start(ca)` event was added in the cache manager workqueue after the `Inv(k)`, the cache manager does not detect the invalidation.