CompuCache: Remote Computable Caching using Spot VMs

Qizhen Zhang*, Philip A. Bernstein§, Daniel S. Berger§, Badrish Chandramouli§,
Vincent Liu*, Boon Thau Loo*
*University of Pennsylvania, §Microsoft Research
*{qizhen, liuv, boonloo}@seas.upenn.edu, §{philbe, daberg, badrishc}@microsoft.com

ABSTRACT

Data management systems are hungry for main memory, and cloud data centers are awash in it. But that memory is not always easily accessible and often too expensive. To bridge this gap, we propose a new cloud service, CompuCache, that allows data-intensive systems to opportunistically offload their in-memory data, and computation over that data, to inexpensive cloud resources. For reduced cost, each cache is hosted by spot virtual machine (VM) instances when possible or provisioned VMs when not. CompuCache provides a byte-array abstraction and stored procedures so users can easily allocate inexpensive caches and specify their behavior. It distributes each stored procedure execution across the instances. In this paper, we discuss challenges in designing the interface, execution strategy, and fault tolerance mechanisms for CompuCache. We propose initial solutions for them, describe types of applications that can benefit from CompuCache, and report on the performance of an initial prototype. It executes 126 million stored procedure invocations per second on one VM with 16 threads.

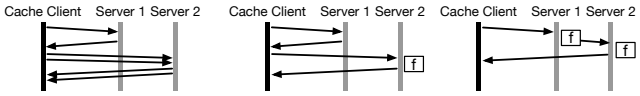
1 INTRODUCTION

Stateful on-line applications require large amounts of data and low-latency compute capacity over that data. Examples are interactive games, fraud detection, stock trading, social networking, and knowledge bases. To meet their latency requirements, these applications are hungry for main memory—if main memory were free, they would use it to cache all their data. Regrettably, memory is not free. Further, even if the application is willing to pay for it, each server has a limited supply.

Ironically, data centers are awash in unused main memory. We estimate that 60% of data center memory is unallocated or unused at any given time (Section 2). One reason is that some of it sits on lightly loaded servers—a natural result of the substantial cluster-level headroom requirements of modern virtual machine (VM) allocators, which need to simplify placement of different-sized VMs and handle periods of peak workload [14]. Machine-level allocation often also requires a degree of headroom to handle the occasional VM resize operation without requiring a VM migration.

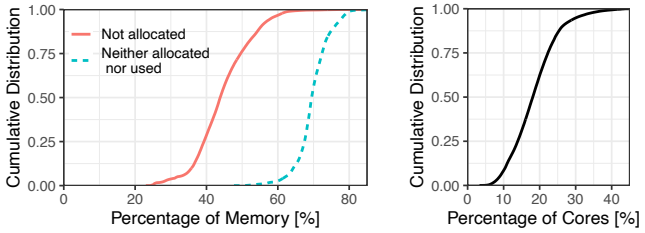
To attract users to make productive use of unallocated resources, cloud vendors offer inexpensive but ephemeral VMs, called *spot instances*. Spot instances can contain both compute and memory resources that applications can access like a normal VM. Further, the high-speed networks in modern data centers make these instances accessible with 100 Gb/s bandwidth and sub-10 µs message latency,

 12th Annual Conference on Innovative Data Systems Research (CIDR '22). January 9-12, 2022, Chaminade, USA.



(a) I/O-only interface. (b) Predefined keys. (c) Ideal interface.

Figure 1: A query that chases pointers between two cache servers, where f is an offloaded function.



(a) Available Memory. (b) Available Cores.

Figure 2: Resource underutilization in Azure clusters. When combining memory not allocated to VMs and allocated-but-unused memory, at least 60% of memory could be used.

assuming the use of kernel-bypass (e.g., DPDK) and CPU-bypass (e.g., RDMA). In exchange for the low price, an application must accept the possibility of losing its spot instance on short notice as soon as the cloud vendor can sell the instance's resources at a higher price.

Leveraging these unallocated resources is, unfortunately, difficult for today's applications. Due to the inherent churn of spot instances, applications may lose work or be unable to satisfy response-time objectives. Further, although a 10 µs network delay approaches that of an I/O bus, it is still 50–100× longer than a local memory access. This may be acceptable for some applications, especially if the accessed data is cached by the caller so that subsequent accesses are local. However, if the application needs to access a large number of records on a spot instance, the number of round trips and resulting latency can quickly add up. For example, if an application wishes to use a spot instance to access a record with pointers to n other records, it costs $n + 1$ round trips to retrieve them all, independent of where the records are located (e.g., see Figure 1a). For these reasons, despite the attraction of spot pricing, many cloud resources sit idle for long periods.

We propose to bridge the gap between applications and the data center's resources by offering a new kind of cache service designed especially for spot instances. The system we propose, CompuCache, is a distributed, remote cache that provides byte addressability and flexible compute pushdown.

We argue that CompuCache must be distributed because, to minimize cost, it has to make do with whatever sized spot instances

are available. Thus, we expect many, if not most deployments of CompuCache to be composed of multiple VMs. If spot instance capacity is insufficient for the application's request, then CompuCache should also use statically provisioned VMs, if they fit within the user's budget. But CompuCache should replace those static VMs with spot VMs as soon as it is cost-effective to do so, taking into account the cost savings of the spot VM, the likely lifetime of the spot VM, the loss of availability while migrating back to a spot VM, the likelihood of getting another provisioned VM when it is needed, etc.

CompuCache must include stored procedures for compute offloading, such as returning all records referenced by a list of record IDs, all graph nodes reachable from a given node in n hops, or the result of a select-project-aggregate query over a table. When a cache spans multiple VMs it is challenging to efficiently support stored procedures, for two reasons. First, an application should be able to control data placement and perform server-side dereferencing. This is difficult for a traditional, sharded key-value API (which is what today's popular caching systems provide) because it requires all accessed keys to be specified when invoking a stored procedure. Chasing n pointers in a record requires at least two round trips: one to read the n pointers in the record on the server and return them to the client, and one to use those pointers to access other records on the server (as in Figure 1b).

Second, having multiple VMs per cache implies that a stored procedure over a cache may require a distributed execution across VMs. The ideal execution strategy may require a stored procedure to chase pointers from one VM to another, as in Figure 1c. The cache API should enable efficient execution strategies like this, while minimizing the complexity of application programming.

A spot instance can be reclaimed at any time, giving its owner only a short time to prepare, typically 30-120 seconds on today's cloud systems. This leads to several challenges: how to migrate the content of a cache that is about to be reclaimed, how to migrate the execution of stored procedures, and how to synchronize the mapping of the logical-to-physical address spaces before and after the migration. It is important to provide precise semantics that makes it easy for applications to cope with these situations. These recovery actions should avoid data loss and minimize any reduction in availability.

A related challenge is how to recover a cache after its VM fails. Some caches are read-only and can be recovered from persistent storage. An updatable cache may need to be replicated and/or have a checkpointing mechanism.

Our main contribution is to propose and explore the design and implementation of this new type of cloud service: a computable cache over spot instances. We discuss challenges in designing its interface, execution strategy, and fault tolerance mechanisms. We propose solutions for those challenges, which we implement in a prototype called CompuCache, and we report on its performance. CompuCache achieves 160 million I/O operations per second using one server VM and 126 million stored procedure invocations per second for offloading a computation that performs aggregation.

The paper is organized as follows. Section 2 presents measurements of the amount of unallocated memory in cloud data centers. Section 3 describes the technical challenges in building CompuCache and how we are solving them. Section 4 presents preliminary measurements of our prototype implementation. Section 5 discusses applications, and Section 6 summarizes related work. Section 7 concludes with open design problems.

2 UNDERUTILIZED CLOUD RESOURCES

Major data center operators report that memory is highly underutilized. Studies of traces from Google [12], Alibaba [4], and Facebook [13] show memory utilization below 50%.

We confirm these results for Azure's large public cloud by characterizing memory usage in 100 compute clusters over a 100-day period. The clusters represent the majority of the fleet and host mainstream internal and external VM workloads. We selected clusters deployed for more than one year and with an average CPU utilization of at least 70%.

We define *unallocated memory* as the fraction of DRAM that the scheduler has not assigned to any VM. Figure 2a shows a CDF of hourly snapshots from the 100 clusters. The median snapshot shows that more than 42% of memory is unallocated, and almost all snapshots have at least 23% of unallocated memory.

We define *unused memory* as the fraction of DRAM that was not touched while it was assigned to a VM. These memory pages can be detected and harvested for CompuCache [17]. Figure 2a shows a CDF of hourly snapshots of the cumulative free memory when considering both unallocated and unused memory. The median snapshot shows that *almost 70% of DRAM is available*, and almost all snapshots show that more than 60% of DRAM is available.

Finally, CPU cores are also underutilized. Figure 2b shows a CDF of the percentage of cores not assigned to VMs or containers. At the median, *more than 18% of CPU cores are available even in our compute-intensive trace.*

Previous measurement studies that report results on underutilized CPU and memory are consistent with our findings [5, 10, 11].

3 OVERVIEW AND DESIGN CHALLENGES

Figure 3 shows the components of CompuCache. Applications of CompuCache run in regular VMs. Examples include database systems, key-value stores, graph databases, and custom data-intensive applications. An application creates and manages caches using a **CompuCache client**, which offers fine-grained control over memory allocation and placement. It uses the **Allocate** function to create a cache with a given capacity C and then uses **Read** and **Write** functions to access any bytes between 0 and $C - 1$.

CompuCache distinguishes itself from existing remote caching systems with two key innovations: It hosts the cache in spot VMs to minimize cost, and it supports efficient compute offloading to speed up application performance.

For the first innovation, CompuCache calls the **cloud VM allocator** to reserve spot VMs to run **CompuCache servers** that in aggregate satisfy the capacity of the cache. As spot VMs are opportunistic resources, they may have different memory capacities and can be reclaimed by the cloud VM allocator at any time. The CompuCache client divides the cache space into fixed-length virtual cache regions (e.g., 1 GB) and assigns the appropriate number of virtual regions to each allocated VM. CompuCache handles spot VM reclamations transparently; when the cloud VM allocator asks

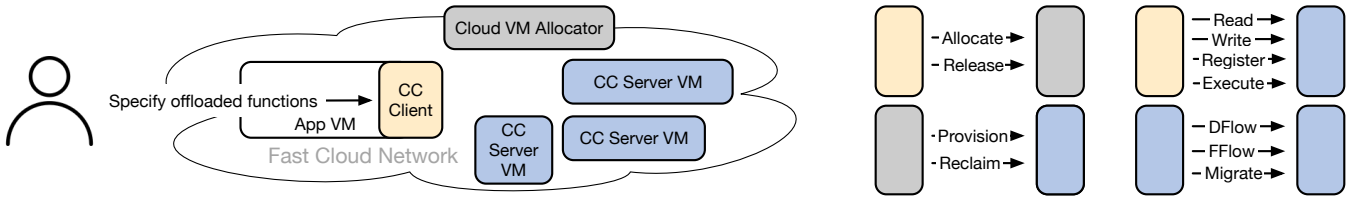


Figure 3: The components in CompuCache (CC) and their interactions.

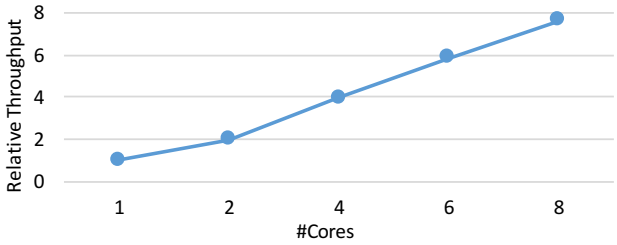


Figure 4: The impact of core count on the executing a sproc that adds the sum of two locations

to reclaim a VM, CompuCache allocates new VMs and migrates the reclaimed VM's data and execution states to the new VMs.

For the second innovation, CompuCache provides APIs that address the limitations of existing remote caching systems, such as the inability of an offloaded function to span VMs or to do server-side pointer chasing (see also Section 5). CompuCache allows its users to implement stored procedures (abbr. *sprocs*) for overall application performance. An application specifies a sproc by calling the **Register** function and executes it by calling the **Execute** function. Sproc code is a parameter to `Register` and is broadcast to all CompuCache servers. On each server, the code is compiled locally as a dynamic library and loaded into the runtime of the server. A CompuCache server might not have all the data needed to execute a sproc, which requires coordination with other CompuCache servers.

We next detail the challenges in the design of the CompuCache interface (3.1), request execution (3.2), and fault tolerance (3.3).

3.1 Interface Challenges

There are several challenges in the interface design to support sprocs with spot VMs as remote cache servers. First, to allocate VMs for hosting a cache, the core count must be specified in addition to memory size. The required number of cores depends on the computational load of each sproc, the mix of different sproc types, the peak throughput, and potential contention between cores. In many cases, the workload varies over time, which implies the core count should too. Another source of complexity is that VMs come in different sizes with different processor generations. It is therefore a significant challenge to choose an initial core count and decide when it should change.

The second challenge is to efficiently support server-side pointer chasing in sprocs, which overcomes a limitation of existing remote caching systems. An application of CompuCache accesses data by its virtual address in the cache, while pointer chasing requires knowing the physical location that a pointer references. CompuCache needs to bridge this gap between virtual and physical addresses without putting much programming burden on users.

The third challenge is presented by the scale-out nature of CompuCache: the accessed data in a sproc can span multiple VMs. A sproc that is executing in one VM may reference data in another VM. This can occur during a sequential reference pattern or when chasing pointers. How to handle such out-of-bounds exceptions and determine the best strategy for continuing the sproc's execution with data on other VMs is challenging.

Sketch of solutions. For the first challenge, the allocation of compute resources, the required initial core count can be chosen by benchmarking the workload mix on one core and assuming it will scale linearly to the peak throughput required. The predicted core count can then be scaled for different VM types based on VM benchmark comparisons. For a read-only cache and a stable workload, we speculate that this will work well enough. Figure 4 shows that the throughput of a CompuCache server running a simple sproc scales up linearly with the number CPU cores, thereby demonstrating there is negligible communication contention between cores. More sophisticated performance modeling will be needed when updates are present to cause contention. If there is significant workload variation over time, then dynamic optimization will be needed.

For the second challenge, server-side pointer chasing, we propose a new data structure, `LocalTranslator`, that provides a `Translate` function:

```
l_addr, l_size ← Translate(c_addr, c_size)
```

where `c_addr` and `c_size` are the cache address and size of a record, and `l_addr` and `l_size` are the server-local (i.e., VM-local) address and size of the record. A sproc invokes this function to map virtual cache addresses into physical server locations. `LocalTranslator` is described in Section 3.2.

For the third challenge, out-of-bounds exceptions are raised by the `Translate` function: if `l_size` equals `c_size`, then the record is fully local; otherwise `Translate` raises an exception. The application has three ways to handle the exception in a sproc. The first is to temporarily bring the data to the current server by calling `DFlow` as follows.

```
l_addr ← DFlow(c_addr, c_size)
```

The second option is to move the sproc execution from the current server to the one with the remaining data using `FFlow` as follows.

```
FFlow(c_addr, c_size)
```

The execution of `DFlow` or `FFlow` incurs data movement between servers, i.e., spot VMs. The difference between them is that `DFlow` moves the *input data* while `FFlow` moves the *current state* of the sproc execution. The final option is to stop the execution and immediately return back to the client. Since the best option depends on the nature of the sproc's computation, CompuCache delegates the decision to the developer of the sproc.

3.2 Execution Challenges

To achieve millions of operations per second, CompuCache must exploit the characteristics of high-speed data center networks and provide efficient support for CompuCache sproc executions.

A critical challenge is a communication transport to make transferring cache I/O and sproc execution requests/responses fast. Sproc execution requests are fixed-size and always small, containing the sproc ID to execute and a few parameters specifying the starting address, response size, and context. In contrast, execution responses vary in size; responses to aggregation requests are small but responses to scan sprocs may be large. Traditional network stacks suffer from overheads that limit both, e.g., with caching issues and repeated context switching.

Another challenge is how CompuCache schedules sprocs to execute on servers. Many sproc execution requests may arrive on a server at the same time. Execution time will vary for the same sproc with different parameters and for different sprocs. Some will complete while others raise out-of-bounds exceptions. Due to this diversity, deciding which sprocs to schedule on which cores is important for optimizing the utilization of each core and load balancing between cores.

The remaining challenges are in how best to support cross-server sproc executions with `DFlow` and `FFlow`. A CompuCache server should construct a `LocalTranslator` for sprocs to chase pointers. When a sproc invokes `DFlow`, CompuCache must know where to find the data and how to bring it to the caller. Similarly, when a sproc invokes `FFlow`, CompuCache must decide where and how to ship the sproc execution.

Sketch of solutions. For the first challenge, CompuCache uses eRPC [15], a user-space RPC library that runs on DPDK [2] or RDMA. DPDK avoids OS kernel overhead by executing the networking in user space. RDMA offers the further benefit of offloading the CPU by running most of the networking protocol on the network interface card (NIC). To leverage the full bandwidth of high-speed networks, CompuCache batches small operations in a single network transfer. This includes batching small I/O requests and responses and *all* sproc execution requests. As response sizes vary, it dynamically decides whether to batch them and what batch size to use.

For the second challenge, scheduling, CompuCache servers spawn one thread per CPU core. A CompuCache client builds a *session* to connect with a server thread. All threads on a server poll I/O and sproc execution requests and process them *in place* by default, i.e., there is no dispatching between server threads. This design avoids inter-thread communication overhead, so CPU cycles are fully utilized for processing requests.

To deal with out-of-bounds exceptions and load imbalance, we introduce a *workqueue* for each server thread. A server thread polls its workqueue for sproc execution requests. It schedules them in the batched order, unless `DFlow` or `FFlow` is called. When `DFlow` is called, the current sproc is scheduled to be inactive and its next request is scheduled to be executed. Inactive requests are executed again when their requested data is fetched. When `FFlow` is called, the current execution immediately terminates.

We propose using a server-wide scheduler. It monitors the utilization of all server threads and can move requests from the workqueues of highly utilized threads to those of less utilized threads.

For the third challenge, CompuCache supports server-side pointer chasing and scale-out execution as follows. When a cache is allocated, the CompuCache client builds a mapping from virtual cache regions to servers based on the list of VMs it receives from the cloud VM allocator. When it connects to a server, it sends the mapping. The server uses the mapping to construct the `LocalTranslator` and to route `DFlow` and `FFlow` requests.

On a server, CompuCache processes `DFlow` and `FFlow` requests by translating them into I/O requests and sproc execution requests. It treats a `DFlow` request as a single read request, which it routes to the right server based on the mapping received from the client. The callback function for this asynchronous read activates the caller sproc. For `FFlow`, after the server terminates the sproc execution, it saves the sproc's current execution context and sends a sproc execution request to the server that hosts the remaining data with the saved context as a parameter.

In the setup of servers, each server builds a single session to every other server. It uses a single server thread to process the requests from all other servers. This design decision is made based on the expectation that most requests come from the client, rather than other servers.

3.3 Fault Tolerance Challenges

Faults are of particular concern in CompuCache—specifically, failures of the CompuCache servers. In addition to traditional types of failures that can affect VMs in a cloud data center, CompuCache servers also need to deal with the eventual reclamation of spot VM instances. The latter is often accompanied by a short grace period.

In contrast to most existing cache solutions, these reclamations introduce a high degree of churn into the membership of VMs comprising the cache. Despite that churn, a sproc execution that accesses data spanning multiple VMs has a need for robust distributed consistency. For example, consider a query over a social graph for the friends-of-friends of a particular user. Assume that execution begins at the server responsible for user A, but A decides to call `DFlow` to access another server B that holds a friend's information. If VM churn leads to the loss of server B before the request is sent out, server A has to be informed of the new server that has the requested data. Similarly, if server A is lost while server B is processing the `DFlow` request, server B needs to know the new destination where it should send the response or whether it should just stop the processing and discard the response.

Complicating the issue of churn is that spot instance sizes are heterogeneous and dependent on the available memory of each server. Thus, when a spot instance is reclaimed, its range can be assigned to multiple replacement instances, which may not adhere to the original computation's view of what is local and what is not.

Sketch of solutions. We consider the fault tolerance for spot VM reclamations and server failures separately, and expect that the former happens more frequently in a deployment of CompuCache.

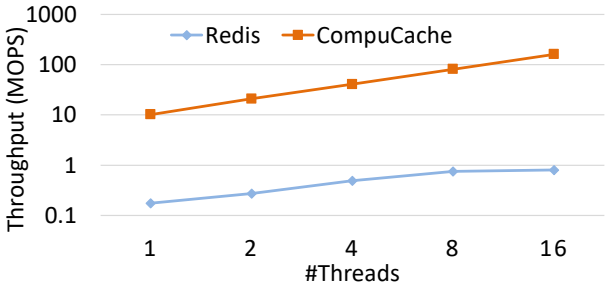
The CompuCache client maintains an up-to-date mapping of all cache regions to CompuCache servers. Recall that the mapping is used to route every CompuCache request to the correct server, to construct `LocalTranslator`, and to process `DFlow` and `FFlow` requests. The client sends the initial mapping to each server during connection setup. On a server reclamation, the client allocates new spot VMs to host the reclaimed cache regions, updates the mapping to reflect the new membership of servers, and broadcasts the membership updates to all servers.

We propose a multi-step data migration protocol for spot VM reclamations that ensures each request to CompuCache *always* goes to the correct server. First, the client assigns the cache regions on the old server to new servers, but does not immediately update the region mapping. It signals each new server to connect to the appropriate old servers to fetch the data in its assigned regions. To minimize the disruption of normal request processing, CompuCache performs migration one region at a time. While it migrates a region, new writes and dependent reads (including sproc execution requests) for that region are paused until the migration is finished. After a region has been migrated, the client updates the region mapping, immediately routes all requests for that region to the new server, and synchronizes the mapping update to all servers. It processes requests to regions that are not currently being migrated without interruption, even if they are on the same server as a region being migrated.

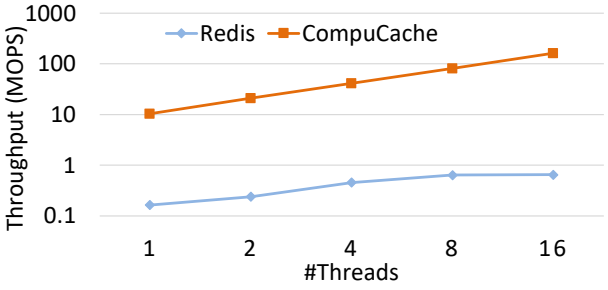
When all regions are migrated, the old server may still be serving read requests and sproc execution requests. In this case, the server quickly drains all read requests. For sproc execution requests, it waits for each thread to finish executing its in-flight sproc request (if any). It then partitions the thread's workqueue into batches, one per server, based on which new server should process each request. It sends each batch to the proper server, if possible combining it with batches from other threads destined for the same server.

During the membership transition, there is a period when servers' views may be stale. This requires special handling of cross-server sproc execution. CompuCache addresses this with tombstones and forwarding addresses. For example, suppose a region r has been migrated from server A to server B, but the client has not yet propagated the mapping change to every server. If server C still has the old mapping, it will send `DFlow` or `FFlow` operations on r to A. To address this issue, A atomically places a tombstone and forwarding address in its local mapping of r, and forwards future `DFlow` and `FFlow` operations on r to B. As an optimization, the `DFlow` and `FFlow` responses can include the forwarding address, which C uses to update its local mapping.

A server failure will cause all requests to regions that the server hosts to eventually time out. CompuCache can create VMs to replace the address space lost by the failed VM. However, it is up to the application to replace its content. For a read-only cache, CompuCache provides a parameter to `Allocate` that maps the cache to a persistent file. For an updatable cache, CompuCache can use traditional methods to ensure durability, such as write-through caching like memcached and Redis, or physical logging and fuzzy checkpointing. During the recovery, the CompuCache client reads the backup data and populates the lost regions. It pauses all other requests to the failed server until recovery finishes, including reads, writes, sproc execution requests, and `DFlow` and `FFlow` operations. Alternatively, CompuCache can avoid data loss due to VM failure by replicating its cache regions [1, 25].



(a) Read throughput.



(b) Write throughput.

Figure 5: I/O performance of CompuCache vs. Redis.

4 PRELIMINARY RESULTS

We now present a preliminary evaluation that demonstrates the performance advantages of CompuCache. We compare CompuCache with Redis [3], a popular in-memory caching system that supports sprocs through the `Eval` function. `Eval` takes as input a Lua script [21] and the keys that are accessed by the script.

Our testbed consists of three Standard_HB60rs VMs in a Microsoft Azure High Performance Computing (HPC) cluster [23] using Mellanox ConnectX-5 NICs and EDR switches running InfiniBand for fast network communications. As Redis requires a traditional TCP/IP network, we configure IP-over-IB for Redis to use the same InfiniBand network as CompuCache.

Cache I/O performance. We first compare the I/O performance of these two caching systems. We measure their throughput to read and write 8-byte records with an equal number of client and server threads. Figure 5a shows the results for reads and 5b for writes. Redis's throughput is 0.2 million operations per second (MOPS) with one thread and peaks at 0.8 MOPS with 8 threads. Thus, Redis does utilize the underlying fast network. In contrast, *CompuCache has 200× higher throughput and scales linearly with the number of threads.* With 16 threads, its throughput is 160 MOPS for both reads and writes.

Sproc execution performance. We next evaluate the performance of executing sproc requests—a key feature of CompuCache. Our first

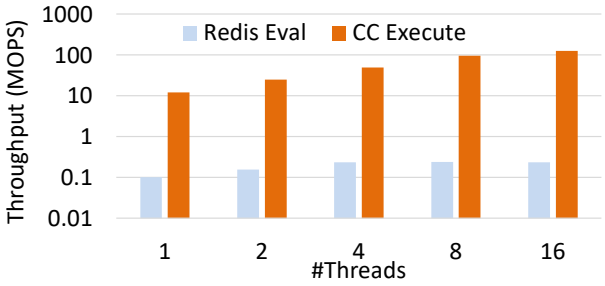


Figure 6: The performance of executing simple sprocs.

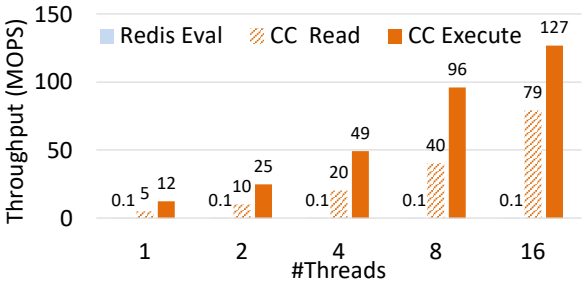


Figure 7: The performance of aggregating two records.

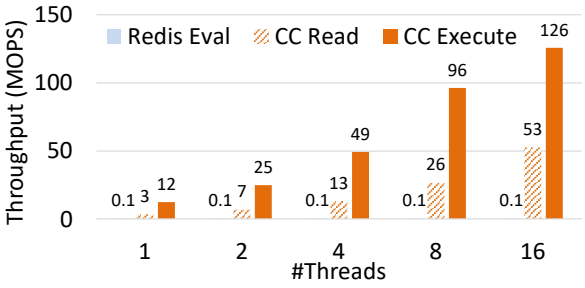


Figure 8: The performance of aggregating three records.

test uses a sproc that simply reads a record and verifies whether its data is correct. Figure 6 shows the throughput of `Execute` in CompuCache and `Eval` in Redis. Here, the performance gap between CompuCache and Redis is even wider. The lower performance of Redis is due to two sources of overhead in executing `Eval` requests: (1) the Lua script is interpreted on the server and (2) a Redis server does not parallelize the Lua runtime. In contrast, CompuCache compiles stored procedures ahead of time and parallelizes sprocs. With one thread, CompuCache can process 12 million `Execute` requests per second, while Redis can only handle 103 thousand `Eval` requests per second. With 16 threads, CompuCache scales by 10× (to 126 MOPS) whereas Redis only scales by 2.3× (to 0.24 MOPS).

As a second scenario, we evaluate an aggregation sproc and control the number of records aggregated per sproc invocation. Aggregating more records means more data movement is saved by executing sprocs, thus demonstrating the advantage of CompuCache over a cache that only supports reads and writes. We compare three cases: (i) CompuCache using sprocs (CC Execute); (ii) CompuCache shipping all data and performing the aggregation

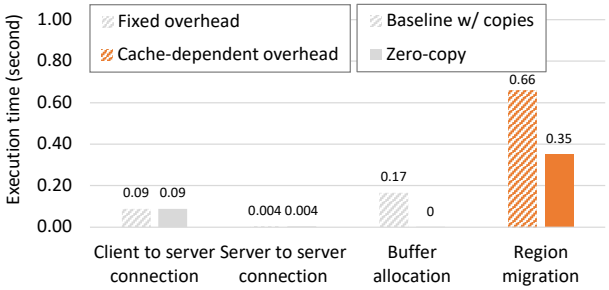


Figure 9: The performance of the components in cache migration and the advantages of zero-copy.

locally (CC Read); and (iii) Redis using `Eval` to push down the same aggregation. Figures 7 and 8 show the results for 2-record and 3-record aggregation sprocs respectively. We find that CC Execute effectively reduces data movement. It outperforms CC Read by 2× and 3× for 2-record and 3-record aggregation, respectively. Redis has orders of magnitude lower throughput than either CC variant due to additional computation and despite reduced data movement.

Cache migration. We now investigate the performance of migrating a cache in CompuCache to handle a spot VM reclamation. Recall from Section 3.3 that it takes several steps to replace a cache server in CompuCache with new spot VMs: the client connects to the new servers, the new servers connect to the old server, and then the regions on the old server are migrated to the new servers. The last step performs the actual data transfers and can be costly. A baseline design is to create eRPC message buffers on both the old and new servers. The old server sends each of its regions to the new server that hosts that region by copying that region's data to its buffer. After eRPC delivers the message to the destination buffer, the new server copies the data to its local region.

Figure 9 shows the performance of the above design when migrating a 1 GB cache to a new server. The figure shows both fixed and cache-dependent components. The time to build connections and allocate buffers is constant regardless of how large the cache is, while the time to transfer the data depends on the cache size. The results show that CompuCache is fast at building connections due to the benefit of fast networks. The overall performance is bottlenecked by buffer allocation and region migration.

We sped up CompuCache cache migration by eliminating the buffer allocation and memory copies in the baseline design with a *zero-copy optimization*. The core idea is to combine memory regions and message buffers. When a cache server allocates a memory region, it also registers the region as a message buffer to eRPC. This design removes the need to allocate message buffers on the fly and to copy memory between CompuCache and its transport. It reduces setup and transfer times for cache migration, which enables a larger cache to be migrated within the time constraint of a spot VM reclamation.

Figure 9 shows that the zero-copy optimization eliminates the buffer allocation overhead and migrates regions 1.9× faster.

Different networks. In addition to the VMs in an Azure HPC cluster, we also evaluate CompuCache performance with general-purpose Azure VMs [22]. We used Standard_D8s_v3 instances, which are cheaper VMs with less network bandwidth (3~4 Gbps

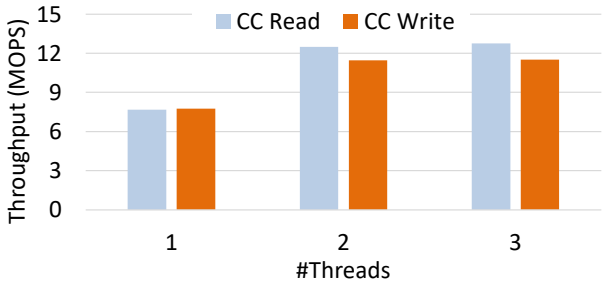


Figure 10: I/O performance in a different network.

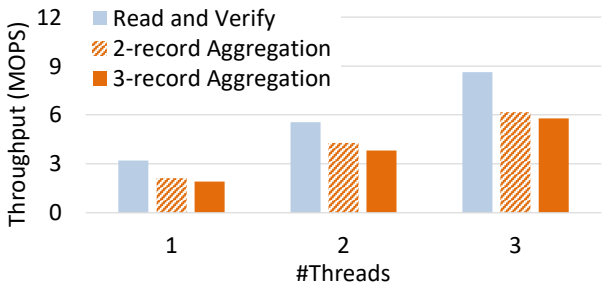


Figure 11: Sproc performance in a different network.

Ethernet vs. 100 Gbps InfiniBand in Azure HPC) and less compute resources (8 vcores vs. 60 vcores in Azure HPC).

Figures 10 and 11 show the throughput of CompuCache executing I/O and sproc requests respectively. Compared to Figure 5, CompuCache can saturate this network with two threads for I/O, achieving 13 MOPS for reads and 12 MOPS for writes.

With less powerful CPUs, the performance of executing sprocs drops correspondingly. With three threads, the throughput is 8.6 MOPS, 6.2 MOPS, and 5.8 MOPS for running the simple read and verify sproc, and 2-record and 3-record aggregations respectively.

This set of experiments shows that the performance of CompuCache adapts to the underlying cloud resources. Even with lower-end VMs, CompuCache still achieves millions of operations per second. It is up to users to make the trade-offs between performance and expense.

Summary. The main takeaway of these experiments is that CompuCache outperforms existing remote caching systems for both single-record I/O's and compute pushdown and that offloading data-intensive operations to the server side allows more performance optimizations. Our evaluation also shows that CompuCache reacts quickly to spot VM reclamation by optimizing the performance of region migration. The evaluation of more benefits from pointer chasing and cross-server sproc execution remains to be done.

5 APPLICATIONS

We briefly survey representative applications that can benefit from CompuCache. These applications have the following characteristics: (1) complex data structures where dereferencing is common when computing queries over data; (2) some churn due to updates to the underlying base data over time; and (3) a need for elastic scaling and use of spot instances to handle transient increase in load.

In-memory key-value store as distributed cache. CompuCache enhances existing key-value stores along two dimensions: it can support parallel compute offloading and can recursively dereference pointers without incurring round trips. The absence of these capabilities can lead to poor performance of some popular key-value stores, such as Redis and Memcached. For example, in Redis, all hash-based and set-based data structures are designed to run on a single machine. By contrast, CompuCache can support a library of parallel data structures backed by a distributed remote cache, with the fringe benefit that it is spot-instance friendly.

Graph databases and knowledge bases. Graphs present a challenging workload due to their scale and inherently stateful operations. When deployed in a parallel setting, sharding requires a single query to span multiple servers. But sharding is rarely perfect, so graph algorithms have to traverse edges across servers.

Using CompuCache, an application can follow edges directly without returning to the client. Many recursive graph queries operate on complex data structures, requiring round trips back to clients. For example, a query to count the number of vertices within k-hops of a particular vertex may involve multiple servers. CompuCache avoids unnecessary round-trips by enabling cross-server pointer chasing. These benefits carry over to knowledge graphs as well, which have a similar data model and topology.

Relational databases. An RDBMS can benefit from using remote memory as an extended buffer pool, semantic cache, and temporary data store. These scenarios are analyzed in detail in [18]. However, they still require the query processor to move a significant amount of data from the remote machine to the one executing the query processor, so it can process the data locally. To avoid this data movement and thereby achieve better RDBMS performance, CompuCache allows data-intensive operations, such as index lookups and predicates on materialized views, to be offloaded to the server that hosts the cache. Using CompuCache, the query processor can easily parallelize the execution of these operations. CompuCache further increases the benefits of offloaded query processing by its ability to use spot VMs for the remote memory servers, thereby reducing cost.

There are still some open questions. For instance, properly sizing VMs to host CompuCache caches requires expertise and accurate predictions of the workload. Another challenge is in join implementation. If a join requires tuples that span multiple VMs, cross-server data shuffling will be needed. This overhead can be alleviated by careful tuning the placement of tuples in cache servers, e.g., partitioning by join keys of frequent join queries. Again, this optimization is workload-dependent.

6 RELATED WORK

In Section 5, we discussed some key differences between CompuCache and in-memory distributed caches such as Redis and Memcached. We briefly summarize other areas of related work.

Near-data processing. A line of work investigates pushing computation closer to the data in distributed databases [6, 9] and disaggregated storage [24, 26, 28]. In all cases, specialized operations can be executed remotely close to the data source to avoid transferring data over the network. CompuCache differs in that it uses spot instances, which have higher degrees of churn, and can support

complex data types where recursive queries across multiple servers are commonplace.

Operator pushdown techniques on smart NICs. There have also been proposals to execute remote operations on programmable smart NICs [16, 19, 20]. Unlike CompuCache, these proposals are limited by the need for specialized hardware. They also focus primarily on operations that are executable only on a single node.

Remote memory for databases. Previous work has proposed novel memory management for DBMSs to utilize remote memories [7, 18]. The ideas differ in that they assume that significant resources are used at dedicated servers, while CompuCache uses opportunistic cloud resources for native compute offloading.

Redy [29] is a new cloud service that we proposed to offer remote stranded memory in cloud data centers as high-performance caches. Redy only allows applications to offload state, while with CompuCache, applications can offload both state and operations over that state.

Memory disaggregation for databases. Separating compute and data with memory disaggregation has been gaining traction for database workloads recently [8, 30, 31, 33]. This new cloud architecture allows DBMSs to scale compute and memory resources independently, but this benefit comes at the cost of data movement between compute and memory. To reduce the cost, recent work [16, 19, 32] proposes to use accelerators, e.g., FPGA and SmartNICs, to offload memory-intensive operators to remote memory servers. Compared to these systems, CompuCache is unique in its capabilities of using opportunistic data center resources and executing distributed functions across multiple servers.

RPC with RDMA. CliqueMap [27] proposes network infrastructure support to make RPCs run efficiently on remote cached data using RDMA. CompuCache differs in its fault-tolerant and cross-server sproc execution designed for spot VMs.

7 CONCLUSION AND NEXT STEPS

We presented CompuCache, a computable caching system designed to maximize the use of unallocated resources in cloud data centers. We described the interface, execution, and fault tolerance challenges associated with its design and implementation.

Next steps are to flesh out our proposed solutions, evaluate the performance of a complete implementation, and validate the implementation's utility for applications. Beyond that, other research questions include integration with the cloud VM allocator, offering user-control over the latency-throughput tradeoff, ensuring optimal distribution of unallocated resources by the users requesting it, and support for the execution and synchronization of updates. There is much to be done.

ACKNOWLEDGMENTS

We thank the anonymous CIDR reviewers for their thoughtful feedback that improved this paper. We also thank Anuj Kalia, Olga Poppe, and Vasileios Zois for their help with eRPC. Qizhen Zhang, Vincent Liu, and Boon Thau Loo were partially supported by NSF grants CNS-2107147, CNS-2104882, ONR N000141812618, and by Google, VMware, and Samsung.

REFERENCES

[1] Apache zookeeper. https://zookeeper.apache.org/.
[2] Data plane development kit (dpdk). https://dpdk.org/.
[3] Redis. https://redis.io/.
[4] Alibaba. Alibaba cluster trace. https://github.com/alibaba/clusterdata.
[5] P. Ambati et al. Providing slos for resource-harvesting vms in cloud platforms. In *OSDI 2020*.
[6] D. F. Bacon and others. Spanner: Becoming a SQL system. In *SIGMOD 2017*.
[7] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *SIGMOD 2015*.
[8] W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang, B. Wang, Y. Wang, H. Sun, Z. Yang, Z. Cheng, S. Chen, J. Wu, W. Hu, J. Zhao, Y. Gao, S. Cai, Y. Zhang, and J. Tong. Polardb serverless: A cloud native database for disaggregated data centers. In *SIGMOD 2021*.
[9] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews. The memsql query optimizer: A modern optimizer for real-time analytics in a distributed database. *Proc. VLDB Endow.*, 9, 2016.
[10] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *SOSP 2017*.
[11] A. Fuerst, S. Novakovic, Goiri, G. I. Chaudhry, S. Prateek, K. Arya, K. Broas, E. Bak, M. Iyigun, and R. Bianchini. Memory-harvesting vms in cloud platforms. In *ASPLOS 2021*.
[12] Google. Google cluster data. https://github.com/google/cluster-data.
[13] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In A. Akella and J. Howell, editors, *NSDI 2017*.
[14] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, et al. Protean:{VM} allocation service at scale. In *OSDI 2020*.
[15] A. Kalia, M. Kaminsky, and D. G. Andersen. Datacenter rpcs can be general and fast. In J. R. Lorch and M. Yu, editors, *NSDI 2019*.
[16] D. Korolija, D. Koutsoukos, K. Keeton, K. Taranov, D. S. Milojicic, and G. Alonso. Farview: Disaggregated memory with operator off-loading for database engines. In *CIDR 2022*.
[17] Lagar-Cavilla et al. Software-defined far memory in warehouse-scale computers. In *ASPLOS 2019*.
[18] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In *SIGMOD 2016*.
[19] F. Liu, C. Barthels, S. Blanas, H. Kimura, and G. Swart. Beyond MPI: new communication interfaces for database systems and data-intensive applications. *SIGMOD Rec.*, 49, 2020.
[20] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartnics using ipipe. In *SIGCOMM 2019*.
[21] Lua. The programming language lua. https://www.lua.org/.
[22] Microsoft. General purpose virtual machine sizes. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-general.
[23] Microsoft Azure. Azure high-performance computing. https://azure.microsoft.com/en-us/solutions/high-performance-computing/.
[24] I. Müller, R. Marroquin, and G. Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *SIGMOD' 20*.
[25] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In G. Gibson and N. Zeldovich, editors, *ATC 2014*.
[26] M. Perron, R. C. Fernandez, D. J. DeWitt, and S. Madden. Starling: A scalable query engine on cloud functions. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *SIGMOD 2020*.
[27] A. Singhvi et al. Cliquemap: productionizing an rma-based distributed caching system. In *SIGCOMM 2021*.
[28] X. Yu, M. Youill, M. E. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker. Pushdowndb: Accelerating a DBMS using S3 computation. In *ICDE 2020*.
[29] Q. Zhang, P. A. Bernstein, D. S. Berger, and B. Chandramouli. Redy: Remote dynamic memory cache. *Proc. VLDB Endow.*, 15, 2022.
[30] Q. Zhang, Y. Cai, S. Angel, V. Liu, A. Chen, and B. T. Loo. Rethinking data management systems for disaggregated data centers. In *CIDR 2020*.
[31] Q. Zhang, Y. Cai, X. Chen, S. Angel, A. Chen, V. Liu, and B. T. Loo. Understanding the effect of data center resource disaggregation on production dbmss. *Proc. VLDB Endow.*, 13, 2020.
[32] Q. Zhang, X. Chen, S. Sankhe, Z. Zheng, K. Zhong, S. Angel, A. Chen, V. Liu, and B. T. Loo. Optimizing data-intensive systems in disaggregated data centers with teleport. In *SIGMOD 2022*.
[33] Y. Zhang, C. Ruan, C. Li, J. Yang, W. Cao, F. Li, B. Wang, J. Fang, Y. Wang, J. Huo, and C. Bi. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proc. VLDB Endow.*, 14, 2021.