Rearchitecting In-Memory Object Stores for Low Latency

Danyang Zhuo
Duke University
danyang@cs.duke.edu

Kaiyuan Zhang
University of Washington
kaiyuanz@cs.washington.edu

Zhuohan Li
University of California, Berkeley
zhuohan@cs.berkeley.edu

Siyuan Zhuang
University of California, Berkeley
siyuan_zhuang@berkeley.edu

Stephanie Wang
University of California, Berkeley
swang@berkeley.edu

Ang Chen
Rice University
angchen@rice.edu

Ion Stoica
University of California, Berkeley
istoica@berkeley.edu

ABSTRACT

Low latency is increasingly critical for modern workloads, to the extent that compute functions are explicitly scheduled to be co-located with their in-memory object stores for faster access. However, the traditional object store architecture mandates that clients interact with the server via inter-process communication (IPC). This poses a significant performance bottleneck for low-latency workloads. Meanwhile, in many important emerging AI workloads, such as parallel tree search and reinforcement learning, all the worker processes accessing the object store belong to a single user.

We design Lightning, an in-memory object store rearchitected for modern, low-latency workloads in a single-user, multi-process setting. Lightning departs from the traditional design by adopting a shared memory model, enabling clients to directly access the object store without IPC boundary. Instead, client isolation is achieved by a novel integration of Intel Memory Protect Keys (MPK) hardware, transaction logging, and formal verification. Our evaluations show that Lightning outperforms state-of-the-art in-memory object stores by up to 9.0x on five standard NoSQL workloads and up to 4.5x in scaling up a Python tree search program. Lightning improves the throughput of a popular reinforcement learning framework that uses an in-memory object store for data sharing by up to 40%.

PVLDB Reference Format:
Danyang Zhuo, Kaiyuan Zhang, Zhuohan Li, Siyuan Zhuang, Stephanie Wang, Ang Chen, Ion Stoica. Rearchitecting In-Memory Object Stores for Low Latency . PVLDB, 15(3): 555 - 568, 2022.
doi:10.14778/3494124.3494138

1 INTRODUCTION

In-memory object stores (e.g., Redis [60], Memcached [49]) are a critical component in data analytics systems. They allow different processes to share state via a simple key-value interface. Traditionally, the in-memory object store is hosted in remote nodes and accessed by clients over the network. This enables ease of state sharing for scalability and replication for fault tolerance. Achieving high performance of in-memory object stores is important for many applications [21, 22, 44].

Emerging workloads, however, are creating a high pressure on the performance of in-memory object stores. To eliminate network overhead, clients are increasingly scheduled to be co-located with their state. This is especially true for emerging AI workloads that a) require intensive communication between multiple workers, and b) are hosted in a single trust domain instead of multi-tenant cloud. For example, in a popular reinforcement learning library RLlib [45], each worker is a separate process running a series of simulations, and a trainer process collects gradients from and pushes updated model to worker processes through an in-memory object store. As long as the reinforcement learning job can fit into a single machine, the underlying framework scheduler [51] will co-locate all these processes. This transforms what used to be network communications to machine-local data transfers for faster training. Instead of using network sockets to communicate with the remote store, many object stores [49, 60] use IPC (e.g., UNIX domain sockets) for low-latency local accesses (Figure 1a). More recent systems (e.g., Arrow [4], Ray [51]) go one step further to satisfy low latency demands—clients only incur IPC overhead for *metadata* operations (e.g., object creation, lookup, deletion); the object data itself is made available by shared memory to avoid copy overhead. Sharing data objects directly via shared memory between worker processes is a suitable choice for many emerging AI workloads. Worker processes belong to the same user, and they are trusted to be non-malicious. As such, shared memory improves performance due to less data copying overheads; it also reduces memory footprint as multiple workers can share data copies.

However, we found that as latency requirements become more stringent, this latter design choice starts to show its bottlenecks. This is because IPC overheads become a performance bottleneck when metadata operations are frequent. Consider the same reinforcement learning example: in each training round, each worker process fetches the current round of the model, runs simulation, and creates a new object that contains the current gradient. The trainer process fetches the gradients, computes the model for the next round, and creates a new object that contains the updated model. The latency of metadata operations via IPC (i.e., object lookup and object creation) ends up dominating the training speed. (See §6.4.)

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494138

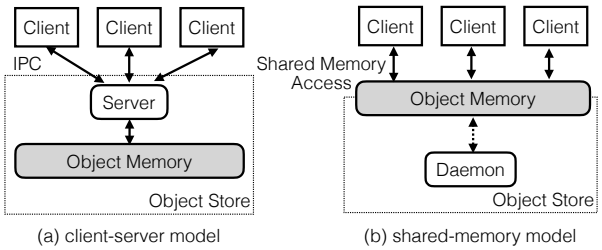


Figure 1: Architecture of In-memory object stores. Existing in-memory object stores (e.g., Redis, Memcached, Plasma) employ the (a) client-server model. Lightning uses the (b) shared-memory model.

The need for IPC stems from the client-server model inherent to today's object stores, as servers mediate most client accesses. Our benchmark shows that, for fetching a 1 KB object in Redis, at least 14 µs out of the 18 µs total latency is due to IPC. (See §2.2.) Even though these workloads run in a single-user setting, the client-server model is important for isolating buggy clients from causing catastrophic failures, e.g., accidentally corrupting object metadata. For instance, RL workers heavily rely on third-party libraries for game simulation; these libraries are trusted to be non-malicious, but bugs are always common and may accidentally modify arbitrary memory addresses or lead to a crash.

This raises an important question: To better serve these emerging workloads, **How should we design in-memory object stores to achieve low latency in a single-user, multi-process setting?**

In this paper, we explore the feasibility and potential performance gains of a completely different design: rearchitecting the object store to completely remove IPC overheads on client operations by performing both metadata and data operations via shared memory (Figure 1b). This will enable all the client operations to proceed at memory speed.

Although this design point holds much promise for performance, there are several technical challenges that we need to address. In particular, we want to *provide isolation across clients using shared memory without changing the underlying OS kernel*. Our goal is to protect against a buggy client from stomping on other clients' metadata accidentally. More precisely, we ensure two types of isolation guarantees: (1) *metadata integrity* in that clients cannot modify the object store's key data structures, and (2) *metadata consistency* in that a crashed client will not leave the object store in an incoherent state. Moreover, the new design must allow for the same set of typical services, such as access control, garbage collection, object subscription, flexible object schema, in today's object stores.

To this end, we design and implement Lightning, a low-latency in-memory object store that represents a drastic departure from today's designs in its use of a shared-memory model for both data and metadata. Clients directly create and fetch objects on the shared memory through a trusted library. We leverage two techniques to ensure fault isolation. For metadata integrity, we leverage recent hardware advances in modern CPUs—Intel Memory Protection Keys (MPK) in particular—which enforce memory isolation within the same address space with low overhead [32]. Lightning uses MPK to ensure that a client thread can access the metadata of the object store only when the thread is executing functions inside our trusted libraries. The metadata of the object store will thus not be corrupted if a client thread misbehaves (i.e., modifying arbitrary memory addresses) when running outside our trusted library. For metadata consistency, Lightning tolerates crash failures via undo logging to provide transactional semantics. When a client thread enters a function in the trusted library, the client thread writes an undo log. If a client process crashes, the daemon process of the object store wakes up, parses the log of the crashed process, and rolls back the object store to a consistent state that is free of metadata corruption. This allows other clients to continue accessing the object store. (See §4.2.)

To achieve high assurance, Lightning's isolation mechanism is formally verified to ensure correctness. Crash consistency is a very important property and correct implementation of crash consistency is difficult in Lightning and other data store, such as file systems [2, 10, 63] and databases [29]. Verifying crash consistency is a fundamental solution to this problem and has been studied extensively in file systems. However, since an object store, such as Lightning, uses unbounded data structures—e.g., linked lists to implement dynamic memory allocators—applying exhaustive symbolic execution as in file system verification [63] leads to state space explosion. To address this, we separate the verification goal into two steps. We first formally verify the undo log implementation that enforces a transactional abstraction on the shared memory, and then verify that the object store uses MPK and the undo log correctly. This separation allows us to verify the isolation properties with proof automation. (See §4.3.)

We also develop a set of typical object store services on top of our object store. We classify object store services into two categories: control-plane services and data-plane services. We implement control-plane services (e.g., access control) in the daemon process, and we implement the data-plane services (e.g., garbage collection, object subscription, flexible object schema) using client-to-client coordination. (See §4.4.)

Our evaluations show that Lightning outperforms state-of-the-art in-memory object stores, Redis and Plasma, by up to 9.0x for five YCSB workloads [74]. We also use Lightning to scale up a Python Monte Carlo tree search program, and it performs up to 4.5x faster than if Redis is used. Finally, we port a popular reinforcement learning framework, RLlib [45], that originally uses Plasma as its underlying in-memory object store on top of Lightning. Lightning speeds up its workloads by up to 40%. Lightning's key limitations include: Lightning does not work for applications that already use MPK for other purposes; when a client crashes, other clients have to wait for the daemon process to clean up the metadata on the shared memory; and Lightning only provides integrity for metadata but not for data when a client misbehaves. (See §7.)

This paper makes the following contributions:

- Lightning, a redesign of in-memory object store architectures for modern, low-latency workloads using shared memory.
- Metadata protection mechanisms that ensure integrity via the use of Intel MPK hardware and metadata consistency using undo logging.
- A verifier that checks the isolation property of our in-memory object store.
- We demonstrate Lightning's benefits on a set of applications, including five standard NoSQL workloads, a Python

tree search program, and a popular reinforcement learning framework.

2 BACKGROUND

We first describe the use cases and software architecture of in-memory object stores. We then benchmark the latency for existing in-memory object stores, using Redis [60] and Plasma [56] as two classic design points.

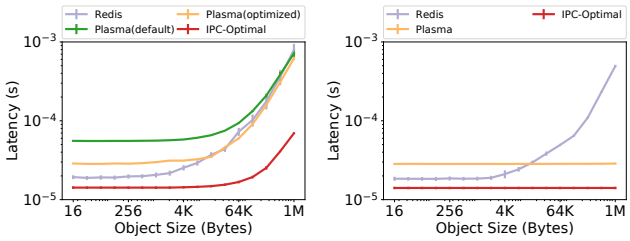
2.1 In-Memory Object Stores

In-memory object stores are a key component for multiprocess applications to share data. Many emerging applications [4, 51] opt to use multiple processes to isolate their workers instead of spawning one thread for each worker in the same address space. This improves software modularity compared to multithreading, and it enables easy fault tolerance [72] and dynamic task invocation (i.e., instantiating new workers from binaries) [79]. Instead of creating many direct process-to-process communication channels, different processes of the same application communicate via a central object store, e.g., Redis, Memcached, or Plasma. Processes share data by creating and fetching objects in the store. This has become a standard programming paradigm due to its simplicity. All the worker processes belong to the same user and are thus trusted to be non-malicious.

Most existing in-memory object stores employ a client-server model, which has a server process to (1) manage object memory and (2) enable object lookup. A typical in-memory object store uses state-of-the-art memory allocators [19, 37] and uses HashMaps to map object identifiers or keys to object buffers. Clients create explicit communication channels to the interactive process in order to issue operations. Unix Domain Socket is often used for fast local client accesses. All the client requests are serialized through the socket, so the server process executes the client requests in order.

There are two options for a client process to access or modify an object buffer. In most traditional designs, such as Redis and Memcached, object buffers transfer through IPC directly. To create an object, the object buffer is sent from the client process to the server. To fetch an object, the object buffer is sent back from the server process to the client process.

In order to further reduce latency, recent designs go one step further: object buffers are in shared memory, and pointer addresses go through IPC. In Plasma, for instance, to create an object, a client process asks the server process to `Create` an object buffer with a given size through IPC. Plasma's server process then allocates a buffer on the shared memory and sends the pointer to the buffer back to the client. After the client process fills the object buffer, it calls a `Seal` function through IPC to allow object lookup for the newly created object. To fetch an object, the client process asks the server process for the pointer to the object buffer through IPC, and the client process directly accesses the object buffer in shared memory. In the second mode, the client process has to communicate with the server process through IPC twice (one for `Create` and one for `Seal`) to fully create an object. Placing objects directly in shared memory is appealing because it eliminates the need to copy data object through IPC. In addition, the total memory footprint reduces because multiple worker processes can share a single copy of data.



(a) Put (b) Get

Figure 2: Redis's and Plasma's client operation latency. The two Plasma curves overlap on (b). Error bars show the standard deviations.

2.2 Key bottleneck: IPC Latency

Although designs like Plasma eliminate data-plane operation IPC overhead, control-plane operations still incur IPC cost. Unfortunately, modern workloads have very stringent latency requirements, making IPC overhead a dominating cost.

We benchmark the latency of Redis and Plasma on an AWS m5-16xlarge virtual machine running Linux 4.15. It has 32 physical cores (Intel Xeon Platinum 8175M CPU, 2.5 GHz) with 256 GB memory. We wrote a latency test tool in C++ for Redis and Plasma. The tool creates, fetches, and deletes objects continuously and outputs the average latency for every 100 operations. The tool uses Redis's and Plasma's default C++ client libraries [33, 57] to access them. For both Redis and Plasma, we configure the tool to issue requests through Unix Domain Sockets. We benchmark the latency for "Put", i.e., object creation, and "Get", i.e., object fetching.

Redis and Plasma are only two instances of in-memory object stores, and in the future, more optimizations can be applied to reduce their latency. To understand the lowest possible latency for these design points, we consider an optimal model for in-memory object store that process requests over IPC. In the optimal model, the latency of "Put" is the round trip latency of sending 1 byte over domain socket plus the latency of memory copy of object size of data. For "Get", the optimal latency is the round trip latency of sending 1 byte over domain socket, assuming data objects are already in shared memory. Figure 2 shows the result.

Redis almost achieves the best possible performance given its design of sending object buffer over IPC. When object sizes are smaller than 2 KB, Redis achieves 18 µs latency for fetching objects. The round trip time of sending 1-byte message over a Domain Socket already takes 14 µs. This means that for small objects, client operations are completely bottlenecked by the latency of IPC. Worse still, this is an underestimate for IPC's performance implication because a typical in-memory object store requires serialization and deserialization on both the client and the server side in order to communicate through IPC [1], which can result in an additional memory copy latency at both the client and the server.

Plasma has better object fetching latency when the object size is more than 64 KB because clients can access the object buffer through the shared memory. However, Plasma suffers from a fixed shared memory setup cost, making its latency high for small objects. Also, because creating an object in Plasma requires communication through IPC twice, its latency is also high (55 µs). To address this issue, Plasma has an optimization that allows sending object buffer

over IPC to create an object. This enables creating an object with a single IPC. We measure its performance and show it in Figure 2a. It speeds up the object creation by 47-49% for small objects (size < 2 KB). Following Plasma evaluations have this optimization enabled.

Batching is another optimization to reduce the impact of IPC overheads between the client and the object store server process. Both Redis and Plasma support batching for object creation and fetching. However, whether batching can improve throughput highly depends on application design and application workloads. Batching cannot reduce client operation latency.

There are certainly many opportunities to optimize Redis and Plasma to approach the optimal model's performance. However, a key question we want to ask in this paper is whether we can build an in-memory object store that is an order-of-magnitude faster than this optimal model for fetching objects and creating small objects.

One potential approach is to maintain the same client-server model and use shared-memory as an IPC channel [3, 8, 35, 39] between the client and the storage process. However, this would require active polling on the shared memory to detect and fetch messages on both ends. This design introduces large amount of unnecessary CPU overheads (evaluated in Figure 6).

3 OVERVIEW

We design Lightning to be a fast in-memory object store for local client accesses. We require Lightning to: (1) *run at memory speed*, i.e., creation, fetching, and deletion should operate without IPC overheads (e.g., sending buffers over a domain socket, signals, or other types of system calls); (2) *isolate misbehaving (non-malicious) clients*, i.e., a misbehaving client cannot accidentally corrupt Lightning's metadata to prevent other clients from making progress, leak memory, or corrupt object lookups; (3) *provide a rich set of object store services*, i.e., the object store provides access control, garbage collection, object subscription, and flexible object schema.

Similar as other in-memory object stores (e.g., Redis [60], Memcached [49], Plasma [56]), Lightning exposes a key-value interface with API shown in Table 1. Lightning maintains a mapping between an `ObjectID` to a `Buffer`. To create an object, a client calls `Create` to create a buffer. The client can modify the buffer and then `Seal` the object. Once an object is sealed, other processes are able to fetch the object through `Get`, which returns a `Buffer`. Objects are also reference counted—a `Get` increments and a `Release` decrements the count. When reference count drops to zero, the object is deleted. Lightning also provides an explicit `Delete` function to delete objects directly.

Lightning's high-level idea is to expose the metadata of an in-memory object store entirely to the client processes in shared memory, ensuring memory-speed access from clients to the object store. On the other hand, the key challenge of this design is to ensure fault isolation while providing a rich set of object store services.

Figure 3 shows Lightning's architecture. The object store exposes its object lookup table and object memory in a shared memory region with all the clients. In addition, each client shares a per-client log with the object store. Lightning object store includes a daemon process that sets up the shared memory and monitors the liveness of its clients. *LightningLib* is the user library that client processes use to access the object store.

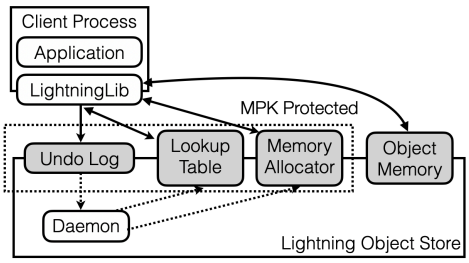


Figure 3: Lightning's architecture. Gray rounded boxes are in the shared memory. The undo log, object lookup table, and the state for the memory allocator are protected by MPK. The client process can access the shared memory through our trusted library, LightningLib. If a client crashes, Lightning's daemon process fetches the client's undo log and rollbacks any modifications in the log to restore the metadata to a consistent state.

Key challenge: metadata protection. Lightning's key challenge is to ensure fault isolation. In particular, we consider three types of faults: (1) a buggy client that contains one or more misbehaving threads that corrupt its memory executing code outside LightningLib; (2) a thread that crashes in the midst of LightningLib execution due to another buggy thread in the same process; (3) LightningLib and the daemon process have bugs so that they do not enforce isolation correctly. We use Intel MPK, transactional logging, and software verification to address these faults accordingly. Our guarantee is that the metadata of Lightning has both integrity (i.e., metadata can only modified by code inside LightningLib) and consistency (i.e., transactional metadata modifications).

Technique #1: Intel MPK. Without protection, a buggy client can accidentally modify the metadata of the object store to destroy its key data structures for memory allocation or object lookup. Lightning uses MPK as a building block for metadata integrity, which is available on all Intel CPUs since Skylake. At a high level, MPK allows a user process to change the permission of a set of memory pages inside its address space, delivering two benefits compared to traditional page tables. First, permission changes are fast. A user process uses a special non-privileged x86 instruction `WRPKRU` to change permission, eliminating context switches into the kernel. Defining the set of pages requires OS support (through the `pkey_mprotect` system call), but once a page set is defined, only one invocation of `WRPKRU` is enough to change permission for all the pages in the set. Second, MPK's protection is at a per-thread granularity. For a multi-threaded process, even if one thread has access to a particular page set, other threads may not have the access to the same page set. Each process can define up to 16 different keys, which is individually mapped to a set of pages. The permission (i.e., the policy of whether a set of pages is read only, or read/write, or non-accessible) for the 16 keys is stored in a per-hyperthread protection key rights register (`PKRU`). CPU checks whether a memory access is permitted by MPK and raises an exception if access is denied. Note that MPK does not provide control flow integrity. A buggy client can 'jmp' to some memory address that accidentally decoded to a valid MPK operation to disable MPK protection. This problem is outside our threat model, and it can be addressed using existing methods [70].

Table 1: Lightning API.

Core (Memory-Speed) Interfaces:	**Description**
Buffer buffer← Create(ObjectID id, size_t size)	Create an object with a given object id and an object size.
Seal(ObjectID id)	Seal an object to make the object accessible through its object id.
Buffer buffer← Get(ObjectID id)	Get an object buffer from an object id.
Release(ObjectID id)	Decrement the reference count. If reference count is 0, delete the object.
Delete(ObjectID id)	Delete an object with a given object id.
MPut, MGet, MUpdate	Described in §4.4.
Auxiliary Functions:	**Description**
Connect(Password password)	Connect to the object store with a password.
Subscribe(Object id)	Wait until the object with the given object id is created.
Exit()	Disconnect from the object store..

We use Intel MPK to ensure Lightning's metadata integrity, by allocating metadata and data in different page sets. Our assumption is that a client does not use MPK's functionality and thus will not circumvent MPK's memory permits. When a client thread is executing functions outside LightningLib, the set of pages containing the metadata is set to be inaccessible using MPK. When the thread calls into LightningLib, LightningLib immediately switches page permissions to make the metadata accessible. When the thread leaves the function in LightningLib, LightningLib switches the permission back. This mechanism ensures that when a client thread misbehaves, it cannot corrupt Lightning's metadata.

Technique #2: Transaction logging. A buggy client thread can also cause other threads in the same process to crash. This means we need to provide transactional semantics for metadata changes even when a thread crashed in the middle of executing a function in LightningLib. When a client wants to modify the object store's state, it has to record the old states in an undo log. If the client crashes in the middle of an operation (e.g., `Create`, `Delete`), the daemon process of the object store collects the client's log and replays the log in the reverse order to rollback partial operations. To avoid concurrent modification to the object store's state, we use a single spinlock to serialize client operations. Our approach guarantees that if a client crashes in the middle of an operation, the state of the object store rolls back to the state before the operation. The spinlock is protected by MPK (part of the store's metadata), so a misbehaving thread cannot hold it. If a thread crashes while holding the lock, the daemon process will undo the operation too.

Technique #3: Verification. We formally verify that LightningLib and the daemon process can enforce client isolation correctly. We cannot use full-program symbolic execution [52, 53, 63, 75, 78] to verify Lightning's crash-fault isolation, because our object store has a buddy memory allocator and a HashMap which involve unbounded data structures (e.g., linked list). This leads to state-space explosion for symbolic execution. We also choose not to use interactive theorem provers over the entire implementation [10, 11, 13, 27, 80] because it requires excessive manual effort. Instead, we verify the isolation property in two steps. First, we implement and verify an undo log implementation that enforces a transaction abstraction on the shared memory. This abstraction requires object store to explicitly initialize a transaction, issue writes to the shared memory, and then complete the transaction. Access to metadata is enabled during this transaction through MPK, and all the modifications to the metadata in shared memory go through this abstraction. The undo log can always rollback an incomplete transaction given the number of writes to the shared memory does not overflow the size of the log. Second, we verify that every possible code path for every operation (e.g., `Create`, `Seal`, `Delete`) has only one transaction and the maximum possible number of writes to the shared memory in the transaction is bounded by the size of the log through a combination of static and dynamic analysis.

4 THE LIGHTNING OBJECT STORE

4.1 Basic Functionalities

Lightning has two basic building blocks. A memory allocator and a HashMap for object lookup. The state of the memory allocator and the HashMap are in the shared memory, allowing direct access from client processes. We have a single spinlock to protect the memory allocator and the HashMap's metadata.

Because objects have varying sizes, we implement a buddy allocator in which we store a set of linked lists of free memory blocks. Memory blocks in the same linked list have the same block size. Our HashMap for object lookup employs a simple open hashing scheme. Lightning hashes `ObjectID` into one of 65536 hash slots, where each hash slot is a linked list.

To `Create` an object, the client (1) acquires the lock, (2) `Malloc` the object buffer on the shared memory, (3) puts the object metadata into the HashMap, (4) releases the lock, and (5) returns the buffer to the client. After the client process fills in the buffer, the client calls `Seal`, which (1) acquires the lock, (2) modifies the metadata of the object to mark the object as sealed, and (3) releases the lock. Only sealed objects can be queried through `Get`. To fetch an object, a client calls `Get` to get the pointer of the buffer in shared memory.

4.2 Isolating Faulty Clients

Lightning leverages MPK hardware and logging to ensure isolation. We store Lightning's metadata in a fixed number of memory pages and use a single MPK key to map to those pages. The permission to those pages is set to be `INACCESSIBLE` by default. During a client operation that needs to modify the metadata, LightningLib switches the permission to be `READ/WRITE`. Note that this permission is at per-thread granularity: even if one thread is modifying the metadata, other threads of the same process that are not calling into LightningLib still cannot modify the metadata.

To tolerate a client thread crashing in the middle of executing a function in LightningLib, Lightning employs logging to ensure crash fault isolation. Naïvely, we can use redo logging [59] in the same way as how modern file systems and databases ensure crash consistency. In redo logging, all updates to the shared memory in an operation (e.g., `Create`) are written to the log first, which are then applied "atomically" to the shared memory when the operation finishes. Such an approach can incur significant overheads for our in-memory object store. Reading modified state requires searching through the log first and then in shared memory, because the log contains the latest states—this incurs high latency. File systems and databases do not have this issue because they make use of page cache for fetching most updated disk states and thus do not need to read the log on the disk.

Therefore, we use undo logging instead to make reads faster. All modifications to the object store state directly go to the shared memory, and the LightningLib records the old states in the client log. Each log entry has 128 bits: 64 bits for the memory address offset relative to the starting address of the shared memory and the other 64 bits for old value at that memory offset. When Lightning's daemon process detects a client crashing in the middle of an operation, it writes the old values to the corresponding shared memory offsets in the reverse order to rollback the object store state to the beginning of the crashed operation. After that, the daemon process releases the global spinlock that the crashed client holds.

We use a simple logging interface. An operation first calls `init_tx` to instantiate a log that is shared between the object store and the client. `init_tx` also switches MPK permissions to allow the client thread to access Lightning's metadata. The client thread can then call `log_write(shared_memory_offset offset, value new_value)` to modify the state of the shared memory. This function first writes the old value of the shared memory at a certain offset to the log and then writes the new value to the given offset on the shared memory. At the end of an operation, the operation calls `end_tx` to destroy the log and switch access permission off for Lightning's metadata.

4.3 Verification of Isolation Property

We want to have high confidence in the design and implementation of our isolation mechanism. If there are bugs, a misbehaving client can leave the object store in an inconsistent state, preventing other clients from accessing their objects. We prove the following two properties on all the client operations:

Theorem 1. *At the end of every client operation, MPK permission for the metadata is set to be INACCESSIBLE (Integrity).*

Theorem 2. *After a crash in a client operation, Lightning's recovery procedure will result in a metadata state where either all or none of the writes are included (Consistency).*

One option is to use exhaustive symbolic execution for verification [53, 63] as it requires less manual effort compared with using interactive theorem provers like Coq [15]. However, Theorem 2 cannot be directly verified using symbolic execution, since object store data structures (e.g., linked list) lead to path explosion during symbolic execution. Consequently, we break down Theorem 2 into the following three theorems and verify them independently.

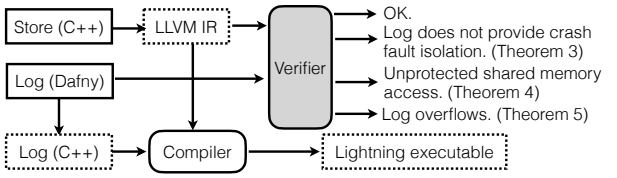


Figure 4: Lightning's verification workflow. Solid rectangular boxes are Lightning's source code, and dotted ones are intermediate representations and the executable of Lightning. The verifier checks for Theorem 3 on the log implementation in Dafny and check for Theorem 4 and Theorem 5 on the LLVM IR of the object store.

Theorem 3. *For up to N write operations, the logging API guarantees that all writes to the shared memory between* `begin_tx` *and* `end_tx` *are atomically applied to the shared memory.*

Theorem 4. *All write operations to the shared memory are within the range of* `begin_tx` *and* `end_tx`. *And each client operation's implementation only uses the logging API to access the shared memory.*

Theorem 5. *The number of write operations in each handler is less than N.*

Figure 4 shows the architecture of our verifier. The high-level workflow is that Theorem 3 ensures the correctness of our log implementation, and Theorem 4 and Theorem 5 guarantee that the log is used correctly in the object store. Note that Theorem 4 implies Theorem 1 because we switch off permissions to the metadata during `end_tx`.

We use Dafny [43] to verify the correctness of the log implementation (Theorem 3). To verify the implementation, we need to explore all possible crash states (i.e., the state of the shared memory when the client crashes). To model all possible crash states during the execution, we use a "countdown" counter, which denotes the number of shared memory and log accesses a client can execute before it crashes. Every time the log implementation accesses shared memory or the log memory, the counter decrements by one. When the counter reaches zero, it means the client operation crashes at this instruction, and all subsequent updates to shared memory and log are thus silently ignored. Therefore, when initializing the counter with a positive symbolic value, we model arbitrary crash behavior during the execution. With this crash model, we verify that the following invariants always hold:

Invariant 1. *If a memory location exists in the log, its first appearance of the log records the original value of the memory location when* `begin_tx` *is called.*

Invariant 2. *If a memory location never appears in the log, its content is the same as when* `begin_tx` *is called.*

We then verify that running the recovery routine, which traverses the log in the reverse order and performs rollback on each entry, on a state where both of the invariants hold, results in same memory state as that at the beginning of the transaction. Note that we only verify the correctness of the logging API up to a certain amount of write operations (N operations), which is bounded by the size of the log itself. Combining with Theorem 5, it is sufficient to

show that the logging API provides correct transactional semantics when used by the client operations.

In order to verify Theorem 4 and Theorem 5, we perform symbolic execution over all the client operations in Lightning. During the execution, we maintain a symbolic counter that tracks the number of log writes issued, the implementation of logging API is replaced with an abstract model that increments the counter by one. To further mitigate path explosion, symbolic execution of the operations is performed in a modular fashion. Instead of inlining all the code when encountering a function call, for a function that contains unbounded loops or recursion, we create a profile for the function manually. The profile includes two pieces of information: the upper bound on the number of log write operations and a predicate on the execution state. Having a state predicate helps us reason about functions whose number of log operations depends on other states such as the calling arguments. For example, a function that contains a loop whose number of iterations depends on its argument cannot be verified without specifying the bound of arguments. When a function F is invoked during the symbolic execution of another function G, the verifier checks if the predicate of F holds on the current symbolic state, if so, it uses the upper bound from the profile of F instead of symbolically executing the function implementation. This modular method makes it possible to verify Theorem 5 without experiencing path explosion. During the verification of each function, the verifier also performs pointer analysis to make sure that access to shared memory is always performed through the logging API. This property, combined with the fact that `begin_tx` and `end_tx` are only called once at the beginning and the end of each handler (the verifier also enforces this during the symbolic execution), is sufficient to prove Theorem 4.

Our trusted computing base (TCB) for verification includes the decomposition from Theorem 2 into Theorem 3, Theorem 4 and Theorem 5, the code compilation process for both Dafny and LLVM IR, the manual translation from Dafny to C++ for our log implementation, and the correctness of all the verification tools we use (Dafny and our verifier). The function profiles are not trusted, because their correctness are checked by our verifier on a per-function basis using symbolic execution.

4.4 Object Store Services

A typical in-memory object store provides a set of object store services that are critical for security, memory efficiency, and convenience. The research question here is whether they can co-exist with our shared-memory model for in-memory object store.

We classify the services into two categories: control-plane and data-plane services. Control-plane services are activated when a client connects or exits, and data-plane services on client operations. For control-plane services, Lightning implements them in the daemon process. The daemon process intercepts the initial connection and exit of clients. Lightning employs a client-to-client model for data-plane services. We implement the following four typical object services in Lightning.

Control-plane: Access control. Lightning enforces access control to prevent unauthorized clients from accessing the object store. Clients cannot directly open shared memory through the POSIX API, i.e., `shm_open`, to get access to the object store. Lightning requires a password for connection establishment. The daemon process accepts connections via a Unix Domain Socket, where clients provide their passwords and the daemon process verifies. Upon a successful connection, LightningLib receives a file descriptor from the daemon process and mmaps it to get access to the shared memory of the object store.

Control- and Data-plane: Garbage collection. If no client crashes, reference counting on a per-object level ensures zero memory leak; this effectively collects garbage in the data plane. In the control-plane, when a client exits (either intentionally or due to bugs), rolling back the object store's state to be a consistent one is not enough. For example, a client can create an object then crash, and thus no other processes know the existence of the object, and the object still occupies the memory of the object store. This means when a client crashes, we need to know what objects are currently referenced in the client process in order to decrement their reference counts.

In the client's log, we dedicate a fraction of its space to persist across different operations. This space contains a HashMap that records the current opened objects, i.e., the client `Create` or `Get` the object without `Release` it. When the client exits intentionally, the client `Release` all these objects. If the client crashes, the daemon process rollbacks the state of the object store and then `Release` all the opened objects for the client. Modification to this HashMap is crash-safe because it is also protected by our undo logs (§4.2).

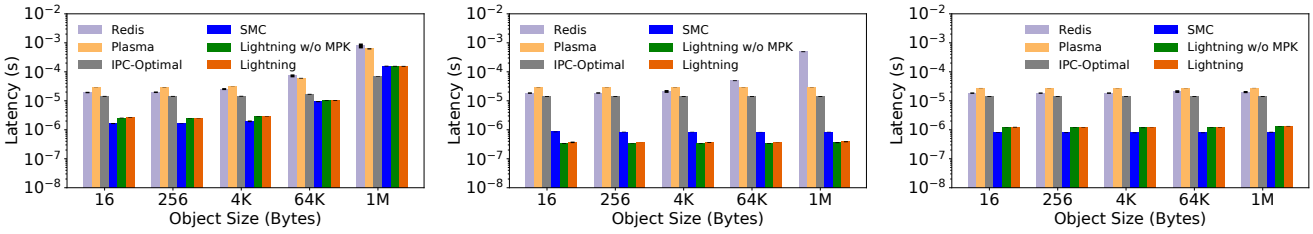
Data-plane: Object subscription. To prevent a client from busy-waiting on an object that has not been created or sealed yet, an in-memory object store usually provides an object subscription mechanism.

Lightning provides a `Subscribe` API to wait until an object is ready (i.e., created and sealed). When a client calls `Subscribe`, it creates a per-object semaphore (`sem_init`) and waits on the semaphore (`sem_wait`). If the semaphore is already created by other clients, the client directly waits on the created semaphore. In this way, the subscribed client is put to sleep by the operating system. When a client calls `Seal` to allow other clients to access the object, the client post (`sem_post`) to the semaphores to unblock all the subscribes for the object and then delete the semaphore (`sem_destroy`). If an object is not subscribed to, none of the semaphore mechanism is triggered.

Data-plane: Flexible object schema. Flexible object schema enables structured data in the object store. Lightning supports user-defined object schema by allowing an object to contain a field-to-value mapping. We provide three additional API for object schema, which are similar to Redis's `HGet` and `HSet` interface:

```
(1) MPut(object_id, fields, values)
(2) MGet(object_id, fields)
(3) MUpdate(object_id, fields, values)
```

When a user creates an object with schema using `MPut`, Lightning serializes the field to value mapping to a customized object format that allows fast lookups through fields. A user can use `MGet` to select the set of fields to query and `MUpdate` specific fields of an object. This feature allows Lightning to support NoSQL workloads, such as YCSB [74].



(a) Put (b) Get (c) Delete

Figure 5: Latency comparison of Redis, Plasma, and Lightning. Error bars show the standard deviations.

5 IMPLEMENTATION

The core functions of Lightning are implemented using around 1800 lines of C++ and 550 lines of Dafny. We manually translate the log implementation in Dafny to C++ in order to generate executable. We implement clients (i.e., `LightningLib`) for C++, Java, and Python to integrate with our client applications (e.g., YCSB [74], RLlib [45], mctspy [48]). We use Java Native Interface to implement the Java client, and Cython for the Python client.

The verifier for Lightning is implemented using around 4500 lines of C++. The verifier generates the LLVM [42] intermediate representation using `clang`, and it accesses the abstract syntax tree of the object store using C++ LLVM library.

Share memory is created using Unix Shared Memory Objects (i.e., `shm_open`, `shm_unlink`). File descriptors that point to the shared memory are passed between the daemon process and clients through `sendmsg`.

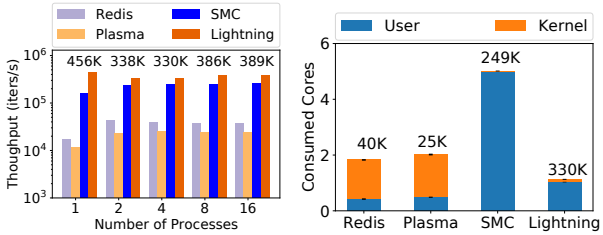
The daemon process wakes up every second to check the liveness of all the clients. It also detects client crashes by monitoring whether a client's process identifier (PID) is alive in the operating system. Linux generates PIDs in a monotonically increasing manner, so we assume there is no PID collision. If a client crashes, the daemon process initiates the crash recovery and garbage collection.

6 EVALUATION

We first perform a set of microbenchmarks, and then evaluate Lightning on five YCSB workloads [74], followed by two AI applications (Monte Carlo tree search [48], and RLLib [45]). Our testbed setup is the same as §2.2.

6.1 Microbenchmarks

Latency. As Figure 5 shows, Lightning has significantly lower latency than Redis and Plasma. Lightning does not have an explicit `Put` API. For Lightning, `Put` means the client calls `Create`, memory copies data into Lightning, and then calls `Seal`. For creating small (< 2 KB) objects, Lightning's latency is 3 µs, much faster than Redis (19 µs) and Plasma (30 µs). For large objects, all systems have similar performance, as memory copy from the client to the object store dominates. For fetching small objects (< 2 KB), the latency of Lightning is 360 ns, nearly two orders of magnitude faster than Redis (18 µs) and Plasma (31 µs). Redis's `Get` latency grows with the object size because Redis transfers objects via IPC; both Plasma and Lightning support zero-copy `Get` so their latency stays constant. For object deletion, Lightning is consistently 15-22x faster than both Redis and Plasma. We also compare Lightning with our optimal

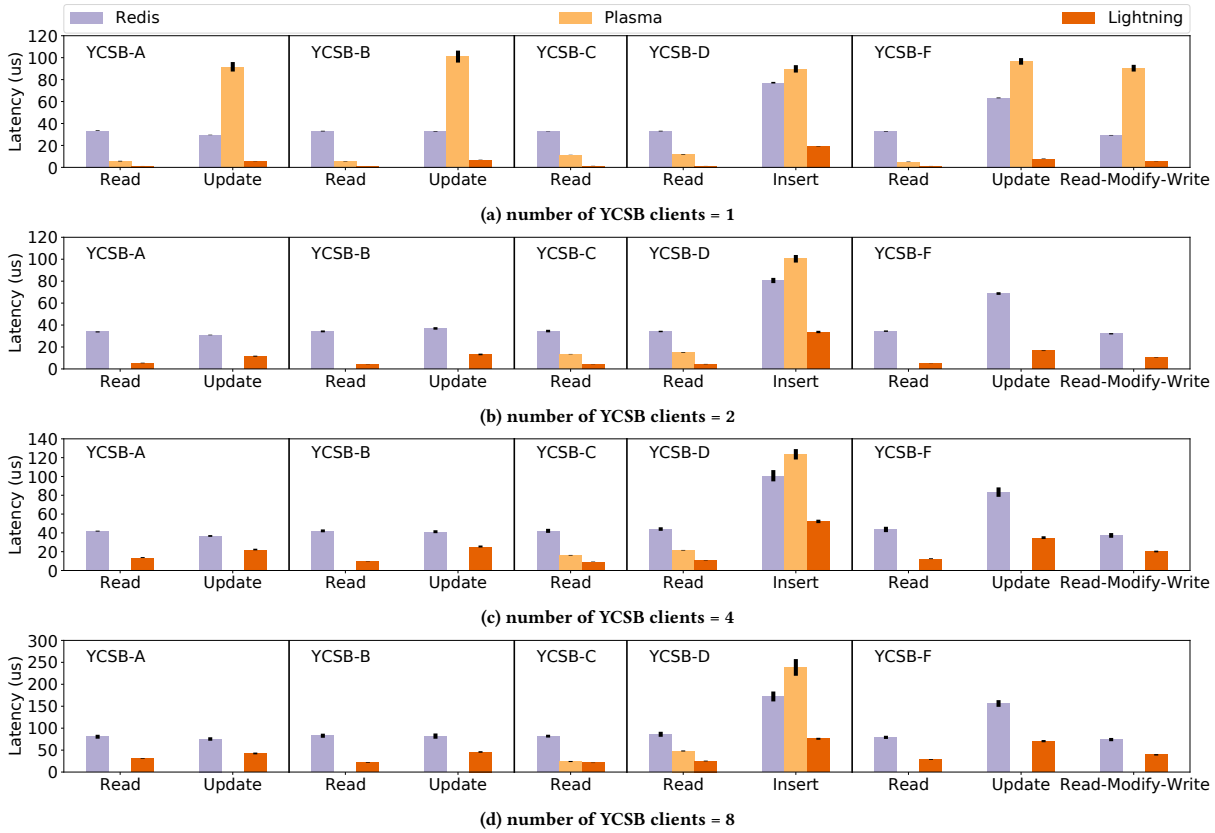


(a) Throughput (b) CPU

Figure 6: Lightning's overall throughput and CPU utilization compared with Redis and Plasma with various numbers of client processes. The overall throughput is calculated by adding up the average throughput of each client process over time. Error bars in (b) show the standard deviations.

performance model for IPC-based object stores as an upper-bound of their achievable performance (§2.2). Here we assume the best latency of `Delete` for IPC-based object store is the round trip latency of sending 1 byte over domain socket. Lightning outperforms this performance model by at least one order of magnitude for creating small objects, fetching objects, and deleting objects. Lightning can achieve this level of performance because the client operations do not need to trigger OS kernel anymore, and they are applied directly by LightningLib on the shared memory. The additional latency due to switching MPK permissions is negligible, which is consistent with the literature [55]. We also implement a version of Lightning that uses shared memory as an IPC channel, where both clients and the server process are busy polling on the shared memory to receive messages. Data objects are still in shared memory. Messages include client requests and responses that contain pointer to objects in shared memory. We use SMC (shared memory channel) to represent this prototype. SMC is significantly faster than both Redis and Plasma, because it also eliminates IPC overheads. However, the busy polling has synchronization overheads between threads (polling on shared memory addresses). Lightning achieves similar performance as SMC. Inserting to and polling from the shared memory channel has overheads in SMC, and ensuring memory ordering in writing the undo log using memory fences has overheads in Lightning.

Throughput. Lightning has high throughput. We use our tool to test the throughput of Lightning, Redis, and Plasma on 1 KB objects (a typical object size in YCSB workloads). We call "put", "get", and then "delete" an object as an iteration, and we test how many



(a) number of YCSB clients = 1

(b) number of YCSB clients = 2

(c) number of YCSB clients = 4

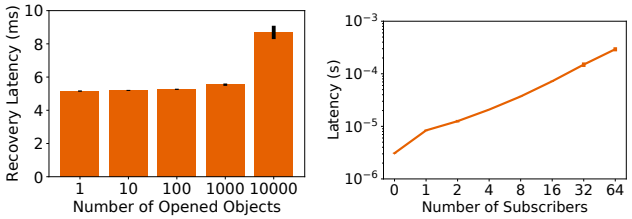
(d) number of YCSB clients = 8

Figure 7: Lightning's latency on various YCSB workloads. Error bars show standard deviations.

iterations clients can achieve. Figure 6a shows the results. Lightning can achieve between 330K and 456K iterations per second. Note here Lightning achieves the highest throughput when there is only one client process. This is because multiple clients have to compete for the spinlock to modify the state of the object store. The lock contention on the shared memory increases client operation latency. In comparison, the maximum throughputs of Redis and Plasma are 44K and 25K iterations per second, respectively. Lightning has 1.3x throughput compared to SMC, because the latter has additional CPU overheads for busy polling.

CPU overhead. Lightning significantly reduces CPU overhead. We measure and break down the CPU utilization using `mpstat` for four clients to continuously "put", "get", and then "delete" an object. Figure 6b shows the result. Redis, Plasma, and Lightning use a similar amount of CPU cores: 1.8, 2.0, and 1.1 virtual cores, respectively. However, their throughputs are 40K, 25K, and 330K, respectively, so Lightning is 14x more CPU efficient. For Redis and Plasma, the majority of the CPU cycles are spent in the operating system kernel for IPC. SMC requires exactly 5 virtual cores because 4 clients and 1 server each needs to do busy polling on the shared memory. SMC can maintain throughput at 249K. This means Lightning is 6x CPU efficient compared to SMC.

Recovery. The speed for recovery depends on the number of open objects. Figure 8a shows the latency for the recovery process for a crashed client with open objects. When the number of opened



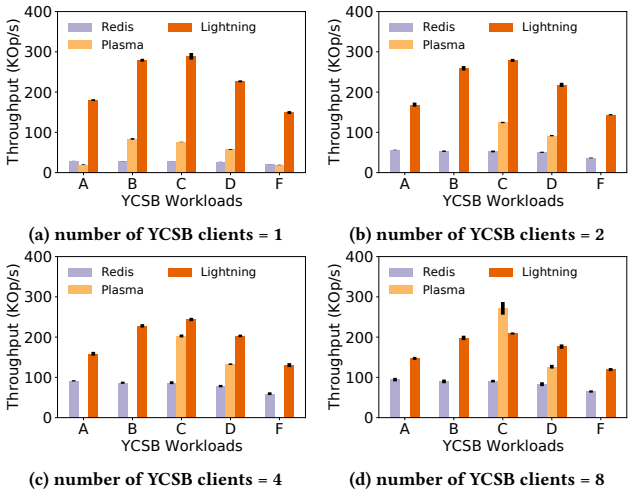
(a) Recovery

(b) Object Subscription

Figure 8: Lightning's latency for (a) crash recovery and (2) object subscription. Error bars show the standard deviations.

object is small (< 100 objects), the recovery takes 5 ms. When the number of opened objects is large, the garbage collection latency is substantial. If the daemon process garbage collects 10000 objects, the entire recovery process is 9 ms.

Sealing subscribed objects. The speed of `Seal` for an object depends on the number of subscribers. Figure 8b shows the latency of object creation when an object has subscribers. When there is no subscriber, the latency is around 3µs. When there is a single subscriber the latency immediately increases to 8 µs. This is because semaphore operations are system calls. When there are more subscribers, the latency increases linearly because in Linux's



(a) number of YCSB clients = 1 (b) number of YCSB clients = 2

(c) number of YCSB clients = 4 (d) number of YCSB clients = 8

Figure 9: Lightning's throughput on five standard YCSB workloads. Error bars show the standard deviations.

semaphore API, `sem_post` only unblocks a single waiting client. To unblock N subscribers, `Seal` has to call `sem_post` N times.

Verification. The verification time has two parts: (1) the Dafny verification of the undo log's implementation and the combination of static and dynamic analysis of the rest of Lightning's source in C++. Verification of the correctness of undo log (Theorem 3) takes 16 seconds. Checking for Theorem 4 and Theorem 5 takes 13 minutes and 35 seconds.

6.2 YCSB Workloads

We evaluate on Lightning with five YCSB workloads [74]. YCSB is implemented in Java, so here we use our Java client to access Lightning. YCSB-A is an update-heavy workload, YCSB-B is read-mostly, YCSB-C is read-only, YCSB-D is read-latest, and YCSB-F is a read-modify-write workload. Lightning cannot support YCSB-E because Lightning does not support range queries on `ObjectID`. Both Redis and Lightning have native support for flexible object schema, so they support YCSB naturally. Plasma does not support flexible object schema, so similar to how YCSB works on memcached, we use Jackson [36] to serialize field-values pairs into a JSON format and then put into an object. In addition, Plasma does not support object update, so we do not evaluate YCSB-A, YCSB-B, and YCSB-F on Plasma when there is more than one client.[1]

Lightning delivers low latency on YCSB workloads. Figure 7 shows the latency comparison with Redis and Plasma. Overall, Lightning improves latency by 1.2-9.0x for different settings, compared with the best of Plasma and Redis for each setting. On a single YCSB client for YCSB-C, Lightning achieves 1 µs read latency compared with 33 µs for Redis and 12 µs for Plasma. Plasma's latency is faster than our microbenchmark. This is because YSCB workloads have locality, and Plasma's client caches object memory addresses in its client library, so subsequent `Get` on the same key is just a client-side lookup. Redis and Lightning do not have this optimization. Lightning performs consistently well on read, update, insert, or read-modify-write across all the measured YCSB workloads.

Lightning significantly improves YCSB workloads' throughput. Figure 9 shows Lightning's throughput in comparison with Redis and Plasma for various numbers of YCSB clients. Lightning achieves 180K, 279K, 289K, 227K, and 149K operations per-second on YCSB-A, YCSB-B, YCSB-C, YCSB-D,and YCSB-F, respectively. Overall, Lightning improves throughput by up to 7.4x for different settings, compared with the best of Plasma and Redis for each setting. Plasma has higher performance than Redis because (1) object data has to be transferred over a socket from the YCSB client to Redis, and (2) Plasma's client caches object memory addresses. Lightning has the best performance when there is only one client because multiple clients can result in lock contention. For example, on the YCSB-C workload, the throughput of Lightning drops from 289K operations per second for a single client to 209K operations per second for 8 clients. On YCSB-C, Plasma actually outperforms Lightning when there are 8 clients. This is because Plasma caches object memory addresses on the client, where each object fetch operation requires the client to grab the global lock in Lightning.

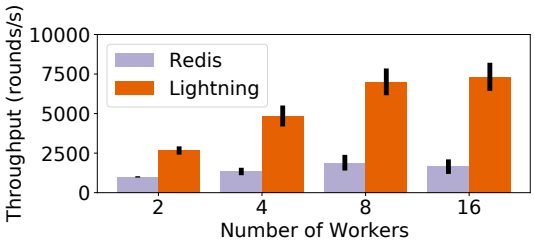


Figure 10: Throughput for Monte Carlo Tree Search. Error bars show the standard deviations.

6.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [16] is a heuristic tree search algorithm, which is proved to be successful in gameplay [24, 28, 65, 66]. For a given input gameplay state, MCTS finds the best move out of a set of moves by repetitively selecting and evaluating search tree nodes and expanding the search tree from the selected nodes.

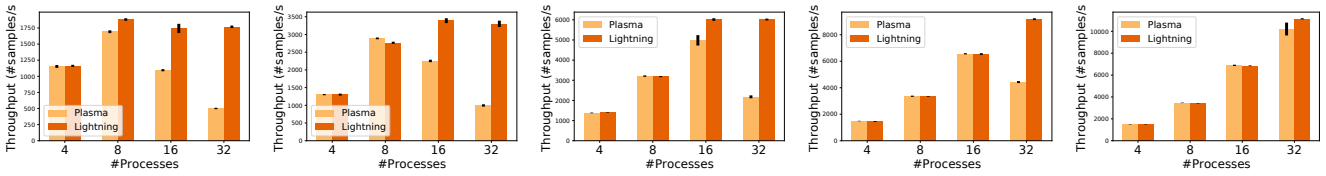
While the vast majority of the AI algorithms are implemented with Python, most MCTS implementations use low-level programming languages like C++ [68]. The reason is that to parallelize MCTS, different parallel workers need to synchronize and update the tree states frequently, which is previously only feasible with threads and shared memory. However, the Python Global Interpreter Lock (GIL) makes it impossible to use threads to scale Python programs and the IPC costs make multi-processing infeasible.

Lightning enables fast object sharing for multi-processing Python applications. We scale a single-thread Python MCTS library mctspy [48] using Lightning with less than 40 lines of code modification compared to the single-thread version. We also scale up the program with Redis as a baseline.[2]

We show the search throughput for Tic-tac-toe in Figure 10. On 16 cores, the search throughput using Lightning as the backend is up to 4.5x higher than the search throughput using Redis. From 2 cores

[1]When there is a single client, we can implement object update via deleting the existing object and creating a new one. However, for multiple clients, because another client can fetch the object between object deletion and creation, this can result in an object-not-found error.

[2]It is difficult to implement parallel MCTS with Plasma because Plasma does not support object update.



(a) Batch size = 5 (b) Batch size = 10 (c) Batch size = 20 (d) Batch size = 40 (e) Batch size = 80

Figure 11: RLLib throughput with Plasma and Lightning with different batch sizes and number of workers. Error bars show the standard deviations.

to 16 cores (4x increase), Lightning scales the search throughput up by 2.7x, while Redis only achieves 1.6x. This is because with Redis, the workers have to communicate with the object store with high-latency IPCs in order to access the search tree.

6.4 Reinforcement Learning

Modern reinforcement learning (RL) algorithms involve deep nesting of irregular distributed computation patterns. The goal of RL algorithms is to learn a policy, which maps the states of an environment to the actions. To learn the policy, the algorithm needs to train the neural networks while interacting with the environments (e.g. a game simulator). Frequent interaction with the unstable external environments makes fault isolation crucial for RL algorithm implementation. The natural heterogeneity of RL algorithms leads to the development of a number of distributed frameworks. One representative distributed framework, Ray [51], is a general task-based distributed system designed for RL workloads and uses Plasma [56] to share objects across worker processes. RLlib [45] further provides abstractions and implementations for various concrete RL algorithms on top of Ray.

We port RLlib on top of Lightning and evaluate its performance with the asynchronous advantage actor critic (A3C) algorithm [50], one of the most widely-used RL algorithms. In A3C, a set of worker processes continuously evaluate a policy network in the external environments and share the resulted gradients with a single trainer process using the object store. Upon receiving the gradients from any worker, the trainer process performs gradient updates on its copy of the policy network, and share the updated network with the worker.

We follow the setting from the experiments in the original paper [50]. We use the convolutional neural network in [50] as the policy network and the Arcade Learning Environment [7] wrapped by OpenAI Gym [9] as the environment for the algorithm, which provides a simulator for Atari 2600 games. We evaluate the performance by the training throughput of the trainer (i.e. number of samples processed per second). Since Atari games do not differ in simulation time, so without loss of generality, we choose Pong as our environment for evaluation. We evaluate RLlib with various numbers of total processes and various batch sizes. Specifically, we test both Lightning and original Plasma implementation with 4, 8, 16, and 32 total processes and batch sizes of 5, 10, 20, 40, and 80.

Figure 11 shows the result. Note that smaller batch size means more frequent synchronization between the trainer process and the workers, and thus communication takes a larger fraction of the total running time. As shown in the figure, when the number of processes is small, Lightning outperforms Plasma by a small fraction, but when the number of processes increases, the performance gaps between Plasma and Lightning become larger. Note that the performance of Plasma decreases while the number of processes increases for small batch sizes, the reason is that in this case the IPC overheads of the Plasma store dominates the training throughput and the overheads increase with the number of total processes. For Lightning, since there is no IPC overhead, the processing speed of the trainer process is the only bottleneck, and thus for small batch sizes, the training throughput for Lightning is roughly the same when the number of processes increases. Comparing the largest training throughput in terms of different numbers of processes between the two stores, Lightning outperforms Plasma by 11% (for batch size 5) to 40% (for batch size 40).

7 DISCUSSION

Isolation model. Lightning adopts common security assumptions in systems of this kind (e.g., Plasma). It ensures fault isolation for buggy clients, but not for actively malicious clients. Such systems target a single-user environment:, where the user spawns multiple worker processes that use the object store. Clients may use `WRPKRU` to switch MPK permissions, but the assumption is that a client process that passes object store access control (§4.4) is not malicious. Future work might generalize systems like Lightning and Plasma to multi-tenant cloud settings. This requires strengthening the use of MPK by static analysis and binary rewriting as done in existing work goal [32, 70]. These enhancements would check that a client never uses MPK primitives to bypass access control outside LightningLib. Lightning provides metadata integrity but not for object data, similar as Plasma [56]. Data integrity can be handled by including object hash values in the metadata region and checking before use. If the client already uses MPK for other purposes, then slight application modification may be required—but this is not a common case for object store workloads.

Latency during client crashes. One fundamental drawback of using Lightning is that, when a client crashes, the rest of the clients have to wait for the daemon process to wake up and clean up the metadata. The latency for the rest of the clients depends on how frequently daemon process wakes up. We currently set the frequency to be once per second, but this can be optimized further with the tradeoff of incurring higher CPU overheads.

Scalability. Lightning serializes client operations using a lock. This prevents multiple clients from operating on the metadata at the same time. We can scale up Lightning by instantiating multiple instances of Lightning, partition the key space using consistent

hashing, and use one instance of Lightning to manage one partition of the key space. But in practice, the current workloads do not require such scaling, as a single thread is already sufficient. We note that our baselines for performance comparison, Redis and Plasma, are both single-threaded applications too.

Remote clients. Lightning significantly outperforms today's in-memory object stores for workloads that co-locate clients and state for low latency. Remote accesses are outside our workload model and can fall back to regular solutions. Lightning does not change how such accesses would happen: the object store can have a set of server threads that mediate operations from remote clients. For such accesses, network overhead is the dominating latency.

8 RELATED WORK

In-memory object stores. In-memory object stores (e.g., Redis [60], Plasma [56], Memcached [49]) are critical infrastructure and have been extensively optimized in the past. Existing work has focused on efficient indexing [14, 22], concurrency [46], speeding up accesses from remote clients [21], replication [76], cache efficiency [71], hardware acceleration [38, 44, 77], tail latency [18], and fault tolerance [54]. Our focus is on low-latency access from local client processes. Our architecture is similar to WhiteDB [73] that both expose their states in shared memory. However, WhiteDB does not provide fault isolation between client processes. Lightning is a drastic redesign of traditional in-memory object store that eliminates client IPC overheads while guaranteeing client isolation. To the best of our knowledge, Lightning is the first in-memory object store that guarantees provable client isolation without IPC overheads due to kernel intervention.

Isolation hardware. Researchers have used Intel SGX [5, 6, 12, 41, 69], ARM TrustedZone [23, 34] to provide isolation for security. Intel MPK is a new hardware feature starting from the Skylake generation, and it has been used to improve the security for several applications. ZoFS [20] is a persistent memory file system that uses MPK to provide isolation between the application and the library to access persistent memory. Poseidon [17] is a memory allocator for persistent memory that uses MPK to protect allocator's metadata. Underbridge [26] and Sung et. al [67] bring memory isolation to microkernels and unikernels, respectively. Hodor [32] uses MPK to build trusted dataplane libraries. Our work is similar to these work that we also use MPK to build our trusted library. However, our goal is different: we build a feature-rich in-memory object store with provable guarantees for metadata protection. We need to deal with crash failure and formal verification of our isolation property. ERIM [70] uses binary analysis and rewriting to prevent MPK circumvention. Libmpk [55] virtualizes MPK so that applications can have an unlimited number of virtualized keys. Our paper is orthogonal to Erim, and we only need to use one key, so we do not need libmpk.

Transactions. Providing transactional semantics via logging is well studied in file systems [10, 61, 63], databases [25, 58], and persistent memory [47, 62], where the goal is to tolerate node crashes by recovering to a consistent state using the log. In contrast, we provide crash fault isolation: when a client crashes, the in-memory object store's state should be consistent, allowing other clients to keep using the object store. In addition, we deal with misbehaving clients that can corrupt memory. Our software architecture is similar to RVM [62] and Rio Vista [47], although the latter projects focus on managing persistent data structures efficiently without considering misbehaving clients—which we address using logging, MPK, and verification. Transactional semantics is also achievable using Intel HTM (Hardware Transactional Memory), although issues around MPK and verification go beyond what HTM can provide.

Crash safety verification. Crash safety can be verified by (1) exhaustive symbolic execution [63] or (2) manual interactive verification [10]. However, neither is appropriate in our problem. Exhaustive symbolic execution is infeasible because object stores have unbounded data structures (e.g., linked lists) for memory allocator and HashMap. Verified file systems often use bitmaps to allocate blocks on the disks. In theory, we can use bitmaps to implement a memory allocator, and symbolic execution would be enough to verify its correctness. However, using a bitmap for memory allocation means the time to allocate a memory block is linearly proportional (in flipping bits in the bitmap) to the number of blocks requested. This is infeasible in high-performance in-memory object stores because in-memory object store has to support low-latency object creation for variable-size objects. Another verification method is to use interactive theorem proving (e.g., using Coq), but it is prohibitively expensive due to the manual verification effort. Our approach is most similar to Nickel [64], a framework to verify information flow control systems, in the breakdown of high-level properties to a set of verification goals for which proof automation can be applied. Our approach does not verify other correctness properties of the object store that existing file system crash-safety verification projects are concerned about. Proving these properties may require manual guidance through annotations [30, 31] or the use of interactive theorem provers [10, 27, 40].

9 CONCLUSION

We design and implement Lightning, a low-latency in-memory object store without IPC overheads for single-user multi-process applications. Lightning allows client processes to directly access the object store's data and metadata, and it enforces fault isolation using Intel MPK hardware, undo logging, and formal verification. We also implement a rich set of typical object store services in Lightning. Our evaluations show that Lightning outperforms state-of-the-art in-memory object stores by to up 9.0x on five standard NoSQL workloads and up to 4.5x in scaling up a Python tree search program. Lightning improves the throughput of a popular reinforcement learning framework that uses an in-memory object store for data sharing by up to 40%. Lightning's source code is available at https://github.com/danyangz/lightning.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their constructive feedback. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Alibaba Group, Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. Danyang Zhuo is supported by an IBM Academic Award and an Amazon Research Award.

REFERENCES

[1] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast Key-Value Stores: An Idea Whose Time Has Come and Gone. In *HotOS*.

[2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2019. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *SOSP*.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*.

[4] arrow 2020. Apache Arrow: Powering In-Memory Analytics. https://github.com/apache/arrow.

[5] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. Speicher: Securing LSM-Based Key-Value Stores Using Shielded Execution. In *FAST*.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*.

[7] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. The Arcade Learning Environment: an Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research* 47 (2013), 253–279.

[8] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1991. User-Level Interprocess Communication for Shared Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 2 (May 1991), 175–198. https://doi.org/10.1145/103720.114701

[9] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *arXiv preprint arXiv:1606.01540* (2016).

[10] Tej Chajed, Haogang Chen, Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich, and Daniel Ziegler. 2017. Certifying a File System Using Crash Hoare Logic: Correctness in the Presence of Crashes. *Commun. ACM* 60, 4 (March 2017), 75–84. https://doi.org/10.1145/3051092

[11] Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. 2018. Verifying Concurrent Software using Movers in {CSPEC}. In *OSDI*.

[12] Chia che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. 2020. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *USENIX Security*.

[13] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a High-Performance Crash-Safe File System using a Tree Specification. In *SOSP*.

[14] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *FAST*.

[15] coq 2020. The Coq Proof Assistant. https://coq.inria.fr/.

[16] Rémi Coulom. 2006. Efficient Selectivity and Backup Operators in Monte-Carlo Tree search. In *International conference on computers and games*. Springer, 72–83.

[17] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. 2020. Poseidon: Safe, Fast and Scalable Persistent Memory Allocator. In *Middleware*.

[18] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *NSDI*.

[19] dlmalloc 2020. A Memory Allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[20] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System.

[21] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *NSDI*.

[22] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*.

[23] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *SOSP*.

[24] Sylvain Gelly and David Silver. 2011. Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. *Artificial Intelligence* 175, 11 (2011), 1856–1875.

[25] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. 1981. The Recovery Manager of the System R Database Manager. *ACM Comput. Surv.* 13, 2 (June 1981), 223–242. https://doi.org/10.1145/356842.356847

[26] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In *USENIX ATC*.

[27] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*.

[28] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. 2014. Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning. In *NeurIPS*.

[29] Theo Haerder and Andreas Reuter. 1983. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.* 15, 4 (Dec. 1983), 287–317. https://doi.org/10.1145/289.291

[30] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *SOSP*.

[31] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *OSDI*.

[32] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX ATC*.

[33] hiredis 2020. Minimalistic C client for Redis >= 1.2. https://github.com/redis/hiredis.

[34] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vTZ: Virtualizing ARM TrustZone. In *USENIX Security*.

[35] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *NSDI*.

[36] jackson 2020. Jackson Project Home. https://github.com/FasterXML/jackson.

[37] jemalloc 2020. jemalloc. http://jemalloc.net/.

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*.

[39] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *NSDI*.

[40] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *SOSP*.

[41] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy Enhanced Secure Object Store. In *EuroSys*.

[42] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[43] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

[44] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *SOSP*.

[45] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *ICML*.

[46] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *NSDI*.

[47] David E. Lowell and Peter M. Chen. 1997. Free Transactions with Rio Vista. In *SOSP*.

[48] mctspy 2020. mctspy. https://pypi.org/project/mctspy/.

[49] memcached 2020. Memcached. https://memcached.org/.

[50] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *ICML*.

[51] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*.

[52] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *OSDI*.

[53] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *SOSP*.

[54] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *SOSP*.

[55] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX ATC*.

[56] plasma 2020. The Plasma In-Memory Object Store. https://arrow.apache.org/docs/python/plasma.html.

[57] plasmaclient 2020. Plasma Client. https://github.com/apache/arrow/blob/master/cpp/src/plasma/client.h.

[58] postgreslog 2020. Chapter 29. Reliability and the Write-Ahead Log. https://www.postgresql.org/docs/9.1/wal-intro.html.

[59] Vijayan Prabhakaran, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2005. Analysis and Evolution of Journaling File Systems.. In *USENIX ATC*.

[60] redis 2020. Redis. https://redis.io/.

[61] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. https://doi.org/10.1145/146941.146943

[62] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. 1993. Lightweight Recoverable Virtual Memory. In *SOSP*.

[63] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *OSDI*.

[64] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *OSDI*.

[65] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (2016), 484.

[66] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go Through Self-play. *Science* 362, 6419 (2018), 1140–1144.

[67] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *VEE*.

[68] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C Lawrence Zitnick. 2017. Elf: An Extensive, Lightweight and Flexible Research Platform for Real-Time Strategy Games. In *NeurIPS*.

[69] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC*.

[70] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security*.

[71] Kefei Wang, Jian Liu, and Feng Chen. 2020. Put an Elephant into a Fridge: Optimizing Cache Efficiency for in-Memory Key-Value Stores. *PVLDB* 13, 9 (May 2020).

[72] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage Stash: Fault Tolerance off the Critical Path. In *SOSP*.

[73] whitedb 2020. Whitedb. http://whitedb.org/.

[74] ycsb 2020. Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/.

[75] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying Software Network Functions with No Verification Expertise. In *SOSP*.

[76] Heng Zhang, Mingkai Dong, and Haibo Chen. 2016. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *FAST*.

[77] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A Case for GPUs to Maximize the Throughput of in-Memory Key-Value Stores. *PVLDB* 8, 11 (July 2015).

[78] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. 2020. Automated Verification of Customizable Middlebox Properties with Gravel. In *NSDI*.

[79] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. 2021. Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems. In *SIGCOMM*.

[80] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. 2019. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *SOSP*.