An Architecture for Memory Centric Active Storage (MCAS)

Daniel G. Waddington
daniel.waddington@ibm.com
IBM Research Almaden

Clem Dickey
dickeycl@us.ibm.com
IBM Research Almaden

Moshik Hershcovitch
moshikh@il.ibm.com
IBM Research Almaden

Sangeetha Seshadri
seshadrs@us.ibm.com
IBM Research Almaden

Abstract

The advent of CPU-attached persistent memory technology, such as Intel's Optane Persistent Memory Modules (PMM), has brought with it new opportunities for storage. In 2018, IBM Research Almaden began investigating and developing a new enterprise-grade storage solution directly aimed at this emerging technology.

MCAS (Memory Centric Active Storage) defines an "evolved" network-attached key-value store that offers both near-data compute and the ability to layer enterprise-grade data management services on shared persistent memory. As a *converged memory-storage tier*, MCAS moves towards eliminating the traditional separation of compute and storage, and thereby unifying the data space.

This paper provides an in-depth review of the MCAS architecture and implementation, as well as general performance results.

1 Introduction

Traditionally, the separation between volatile data in memory and non-volatile data in storage devices (e.g., SSD) has been clear. The interface and semantics between the two domains is well defined; that is, in the event of power-reset or power-fail events, data in memory is lost and, in turn, is then retrieved from storage during the recovery process.

With the advent of Persistent Memory (herein abbreviated to PM), such as Intel's Optane DC Persistent Memory Modules (see Figure 1), this conventional separation of memory and storage begins to blur. Because PM behaves like memory, operations on data held within PM can be performed in-place without having to first load, and potentially de-serialize, from storage. Likewise, data written to PM need not be pushed down to storage to assure its retention. The result is that operations on durable data can be performed at an order-of-magnitude lower latency than has been previously possible. Nevertheless, the Achilles' heel of PM is that data management services traditionally realized by enterprise storage systems (e.g., access control, encryption, replication, compression, versioning, geo-distribution) cannot be easily realized with additional software.

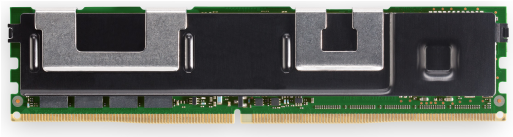


Figure 1: Optane DC Persistent Memory Module

PM raises the data preservation boundary up the stack into the main memory space. It provides non-volatile memory hardware that sits below existing volatile caches but that, unlike existing DRAM-based memory, retains data in the event of failure or reset. The caveat is that data must be explicitly flushed from the CPU cache (e.g. via `clflushopt`) for its persistence to be guaranteed.

It is also "byte-addressable" in that it is directly accessed via load-store instructions provided by the CPU. Intel Optane DC PMM, which uses 3D XPoint (3DXP) technology, operates at a cache-line read-write latency of around 300*ns* (see [1, 7] for more detail). Even though this is slower than DRAM access latencies ($\sim 100ns$) it is at least 30x faster than state-of-the-art storage (e.g., NVMe SSD). Capacity of PM is also about 8x that of DRAM[1].

Another consequence of PM being attached to the system as memory is that it allows use of Direct Memory Access (DMA) and Remote DMA (RDMA) to move data around. For example, data can be copied from PM to the network (via RDMA) or to another device such as a GPU (via DMA), without requiring execution by the CPU. This frees the CPU to perform other tasks rather than executing `memcpy` loops in order to move data. Today, NVIDIA/Mellanox RDMA network adapters can transfer data at near 400Gbps (50GiB/s) and therefore, using multiple adapters, can even keep pace with the performance of PM.

[1] For 3DXP, which is based on lattice-arranged Phase Change Memory (PCM).

1.1 Current Limitations of Intel Optane PM

While Intel Optane PMM provides many useful PM features as just discussed a number of limitations are evident in the current generation.

- *Endurance* - lifetime endurance of the hardware is significantly less than DRAM (3DXP at 10^6 writes, DRAM at 10^{10}) although orders-of-magnitude higher than NAND-flash. For intensive data write operations pushing through the cache this may be a significant limitation [2].
- *Asymmetric Performance Scaling* - write performance does not scale with increasing number of threads, while read performance scales at around 1.2% degradation (from linear) per-core up to 28 cores [13].
- *64-bit Aligned Atomicity* - only aligned 64-bit writes can be guaranteed to happen atomically by the hardware. There is currently no hardware support for multi-write atomicity/transactions and therefore this burden is left to the software.
- *Reliability & Serviceability* - to provide maximum performance DIMMs must be configured to stripe data across 6 devices. In the event of a single DIMM failure, data on all of the DIMMs is effectively lost.
- *Cost* - although current 4Q2020 cost is $\sim$ 0.5x than that of DRAM, it is an order-of-magnitude higher than NAND-flash ($\sim 7\$/GB$ versus $\sim 1\$/GB$)

2 Design Objectives and Solution Positioning

With the previously discussed characteristics of PM in mind, the following tenets in the design of MCAS were made. The solution should:

1. Allow PM h/w resources to be shared as a network-attached capability using RDMA to provide maximum transfer speed. Data sharing across independent nodes should be possible with appropriate locking/serialization provided by the MCAS system.
2. Maintain an immediate consistency model with guaranteed persistence (i.e. data is known to be flushed from volatile caches when a write is made).
3. Support zero-copy (RDMA-only) transfer of large data chunks enabling bulk data movement with CPU `memcpy` execution.
4. Minimize round-trip latency so that small reads/writes can be performed synchronously reducing s/w complexity in the client.

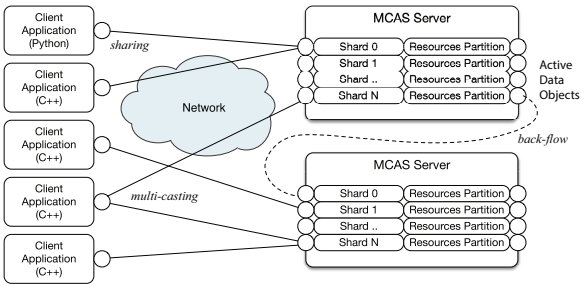


Figure 2: MCAS High-Level Architecture

5. Scale through sharding to bound any performance degradation due to locking.
6. Provide flexible software-defined deployment for both on-premises and cloud. Support both containerized and virtual-machine based deployment scenarios.
7. Enable safe, in-place user-defined operations directly on PM (both general and domain-specific).
8. Provide flexibility in the custom service layering (e.g., combining replication and tiering).

MCAS is positioned as a *converged memory-storage tier* providing high-performance random access to durable data. Because MCAS is based on PM it can provide fine-grained durability (per write) as opposed to snap-shotting. Even with synchronous, consistent and guaranteed-persistent replication across multiple nodes, MCAS can support still millions of updates per second.

3 MCAS Core Architecture

MCAS is implemented as a Linux process (known as the 'shard' process) of which multiple instances can be concurrently deployed on the same machine. An MCAS process instance manages one or more network end-points, each corresponds to a separate *shard* (see Figure 2).

Shards are single-threaded and manage request handling for a *set* of pools. They can be accessed concurrently by multiple clients (from different nodes in the network) and can be grouped into larger virtual data domains through client-side clustering techniques such as consistent hashing [12].

Each shard optionally maintains Active Data Object (ADO) processes that provide custom functionality to the store. The ADO processes themselves may act as clients to other MCAS nodes (back-flow). ADOs are discussed in more detail in Section 4.

Resources (memory, CPU cores) are statically allocated to each shard through the MCAS configuration file. An example two-shard configuration file is shown in Listing 1.

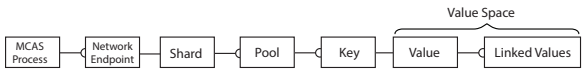


Figure 3: MCAS Entity Relationships

```
{
  "shards" :
  [
    {
      "core" : 0,
      "port" : 11911,
      "net"  : "mlx5_0",
      "default_backend" : "hstore",
      "dax_config" : [{
          "path": "/dev/dax0.0",
          "addr": "0x9000000000" }]
    },
    {
      "core" : 1,
      "port" : 11912,
      "net"  : "mlx5_0",
      "default_backend" : "hstore",
      "dax_config" : [{
          "path": "/dev/dax0.1",
          "addr": "0xA000000000" }]
    }
  ],
  "net_providers" : "verbs"
}
```

Listing 1: Example MCAS two-shard configuration

Each shard serves a single network end-point established using the *libfabric* library, which is part of the Open Fabric Interfaces (OFI) framework[2]. This library provides a common abstraction layer and services for high-performance fabrics such as RDMA verbs, Intel TrueScale, and Cisco VIC. It also includes a provider for plain TCP/IP socket (TCP or UDP) but without user-level and zero-copy capabilities. MCAS primarily supports the RDMA verbs and sockets providers.

Pools are they next level of data collection. Each pool can only belong to a single shard, which in turn means that the handling of operations for a specific key-value pair is always performed by the same shard and thus same thread. Pools represent the *security boundary* from a client perspective. That is, access control and memory resources are all bound to a pool. If a client has access-rights to a pool, then they also have access-rights to all other key-value pairs in the pool.

The overall data entity schema is given in Figure 3.

3.1 Client API

Client applications interact with MCAS by linking to the client-API library (`libcomponent-mcasclient.so`). This library provides a C++ based interface to MCAS. The basic operations are very typical of a traditional key-value store (see Table 1); they operate on *opaque* values that are identified by a unique key. Both keys and values are variable length and there is no restriction on their size.

[2]https://ofiwg.github.io/libfabric/

Function	Description
`create_pool`	Create a new pool or open existing pool
`open_pool`	Open existing pool (optional create on demand)
`close_pool`	Release handle to pool
`delete_pool`	Securely delete pool and release pool memory to shard
`configure_pool`	Configure pool (e.g., add secondary index)
`put`	Write small (< 2MiB) key-value pair. Optionally allow overwrites
`get`	Read small (< 2MiB) key-value pair
`async_put`	Asynchronous version of put
`async_get`	Asynchronous version of get
`free_memory`	Free memory allocated by get call
`erase`	Erase key-value pair from pool
`async_erase`	Asynchronous version of erase
`get_attributes`	Get attributes for pool or key/value pair
`get_statistics`	Get shard statistics
`find`	Search key space in secondary index

Table 1: Basic Client API Summary

3.1.1 Zero-copy Bulk Transfers

MCAS also provides APIs for moving data to and from client-host memory without a memory copy (memcpy) operation being performed under the hood. These *direct transfer* APIs (see Table 2) are realized through the underlying RDMA network hardware and allow data to be moved directly from packet buffers into user-space memory (see Figure 4). The memory for the direct APIs must be allocated (e.g., via POSIX `alloc_aligned`) and then registered with the RDMA stack via the MCAS `register_direct_memory` call. Under the hood, the direct APIs use RDMA read/write operations. However, because the semantics of the MCAS protocol is persistent-on-completion, two-sided operations (i.e. send/recv) are still used to provide the acknowledgments (see Appendix A for detail).

The direct APIs can also be used with NVidia GPU-direct capabilities[3]. This allows data to move from the MCAS server, across the network and then directly from the NIC hardware into an application-defined region of memory allocated inside the GPU. In this scenario, the CPU "host-code" on the client must acquire the region of GPU memory (e.g., via `cuMemAlloc`) and then register this memory with the MCAS `register_direct_memory` call. On completion of the direct call on the CPU, movement of data into or out of the GPU is known to be complete.

Depending on the PCIe arrangement and NIC hardware, direct transfers are able to achieve transfer rates of tens of GiB/s. Of course, these transfers do not require CPU instruction execution and therefore the CPU is free to perform other useful work.

[3]https://docs.nvidia.com/cuda/gpudirect-rdma/index.html

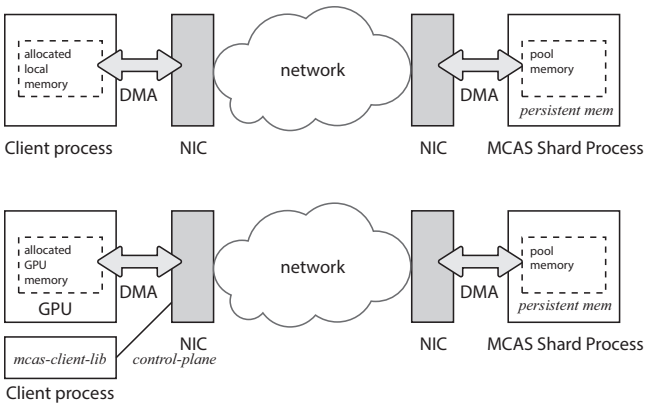


Figure 4: MCAS Direct Transfers

Function	Description
`register_direct_memory`	Register client-allocated memory for direct API use
`unregister_direct_memory`	Unregister client-allocated memory for direct API use
`put_direct`	Zero-copy large put operation using client provided memory
`get_direct`	Zero-copy large get operation using client provided memory
`get_direct_offset`	Read sub-region of pool memory directly
`put_direct_offset`	Write sub-region of pool memory directly
`async_put_direct`	Asynchronous version of put_direct
`async_get_direct`	Asynchronous version of get_direct.
`async_get_direct_offset`	Asynchronous version of get_direct_offset
`async_put_direct_offset`	Asynchronous version of put_direct_offset
`check_async_completion`	Check for asynchronous operation completion

Table 2: Advanced Bulk-transfer API

3.1.2 Direct Offset Operations

For operations on large areas of memory the ability to perform sub-region read/write operations is useful. For this, MCAS provides the `xxx_direct_offset` APIs (see Table 2) These functions allow direct read and write operations for a region of the value space (associated with a key) that is defined by base offset and size pair.

3.2 Primary Index Component

MCAS uses a primary index to manage the mappings from key-to-value and an optional secondary index for scanning of the key-space.

The primary index is provided as a *storage engine* component that provides broader services such as memory management. It is a pluggable component that implements are predefined interface (`IKVStore`). There are currently three storage engines included in MCAS: `hstore`, `hstore-cc` and `mapstore`.

`hstore` uses persistent memory for the main hash table and volatile memory for the memory allocator itself. Alternatively, `hstore-cc` uses a persistent memory based memory allocator. This allocator is slower than its volatile memory counterpart, but it does not require rebuilding after reset.

For DRAM-only scenarios, the `mapstore` backend is available. This is based on a C++ STL `ordered_set`.

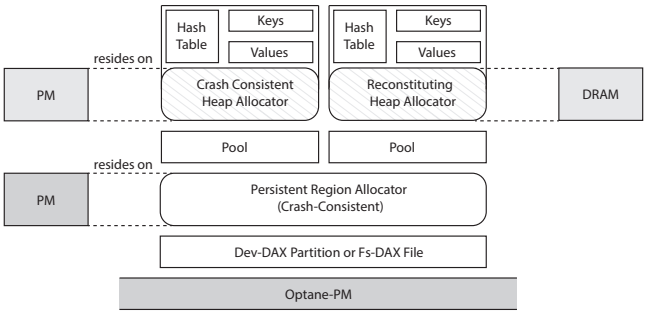


Figure 5: Memory Management Architecture

3.2.1 Memory Management

Figure 5 provides an overview of the `hstore` memory management architecture. At the lowest level, MCAS uses either a *device-DAX* partition, which is a fixed-size partition of a given interleave set or an *fs-DAX* file. Device-DAX is used when On-Demand Paging (ODP) hardware is not available. A device-DAX partition or fs-DAX file is configured for each shard.

To manage the shard memory resources, `hstore` and `hstore-cc` use a coarse-grained crash-consistent heap allocator, known as the *region allocator*. This allocates memory for individual pools. Because shards are inherently single-threaded the region allocator need not be thread-safe and is therefore lock-less. Memory is managed at a 32MiB granularity using an array of region descriptors (offset, length, identifier) that are maintained in persistent memory. Updates to region descriptors are made write-atomic by using a simple undo-log approach.

MCAS has no restrictions[4] on key and value lengths. This means that a heap-allocator is necessary to support variable-length region allocation. To support high-rates of key-value pair insertion and deletion, `hstore` maintains a heap allocator for key-value data in volatile (DRAM) memory. However, because the state of the allocator is neither crash-consistent or power-fail durable, we must "reconstitute" its state after restart. To achieve this, MCAS uses the key-value length information that is stored in PM (and is crash-consistent) to rebuild the allocation state.

[4] Actually, the size is currently restricted by the RDMA verbs maximum frame size, which is 1GiB.

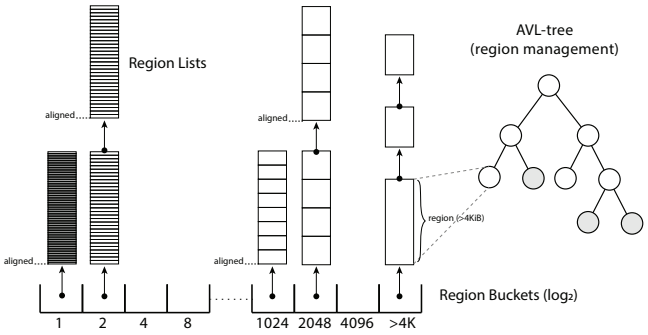


Figure 6: Reconstituting Heap Allocator

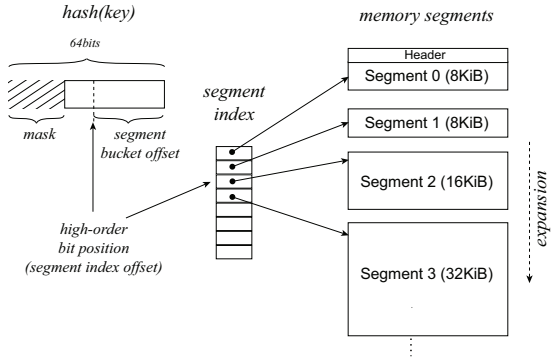


Figure 7: Bucket-segment mapping

The reconstituting allocator manages a set of buckets that represent different 2^n sizes (see Figure 6). Objects (key or value data) up to a given size (4KiB) are allocated in region lists. Each region is 1MiB. Objects are allocated by linear scanning of the regions belonging to the corresponding bucket. If there are no free slots in any region, a new region is allocated. Regions and large-objects (> 4KiB) are allocated using an AVL-tree based allocator. If all slots in a region become free, the region can be returned to the AVL-allocator.

3.2.2 HStore Hash Table

The core of `hstore` is a hopscotch hash table [5]. The hash table is maintained in persistent memory. Expansion of the table is achieved by adding successivevely larger *segments*. Each additional segment doubles the table size. Currently, shrinking of the hash table is not supported.

To manage the mapping between the 64-bit hashed key, we adopt a strategy also used by the Intel TBB hash table implementation [11]; see Figure 7. The basic idea is to partition the hash value into a left hand mask (where bits are ignored) and a right hand set of bits representing the segment index and the segment offset (i.e. designating the bucket). The high-order bit, outside of the mask, is used to indicate the segment index. The remaining bits are used to define the bucket/segment offset.

The hash table memory layout is given in 8. Each entry in the table contains a *hop information* bitmap of H bits (H=63), that indicates which of the next H entries contain items that hashed to the current entry's virtual bucket. In addition to the hop information, each entry contains a state field, key-value pointer-size pairs, and in-lined keys and values (when sufficiently small). Each entry fits a single cache line (64 bytes).

Segments are added by reducing the left hand mask, thus enabling another position for the high-order bit. The segment index is needed because new segments cannot be assured to be allocated in contiguous memory. They are linked together to aid navigation of buckets beyond the segment boundaries.

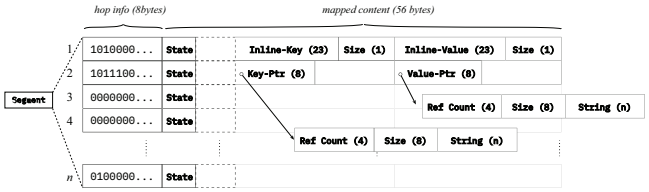


Figure 8: Hash table arrangement

3.3 Secondary Index Component

MCAS also supports the dynamic loading of a non-clustered ordered index on the primary key, which we term the *secondary index*[5]. This index, which is also pluggable, implements a predefined interface (`IKVIndex`). It manages the key space only and its principal function is to provide an index that can be efficiently scanned in key-order. Scanning is based on exact match, prefix or regular expression. Currently, MCAS provides only one (non-volatile) secondary index based on a volatile red-black tree (C++ STL `map`). Nevertheless, any alternative secondary index can be easily developed and integrated into the system.

4 Active Data Objects (ADO)

A key differentiator for MCAS is its ability to perform "push-down" operations in what are termed Active Data Objects (ADO). The ADO mechanism is based on an *open protocol* layering approach (see Figure 9) in which a developer can implement a client-side library (adapter) and server-side plugin that handle and interpret a custom protocol. Together these two components are referred to as the *personality*.

The ADO plugin is statically associated with a shard through a parameter in the MCAS configuration file.

[5] We use this term slightly different from the conventional database interpretation of forming an index on a different key from the primary key.

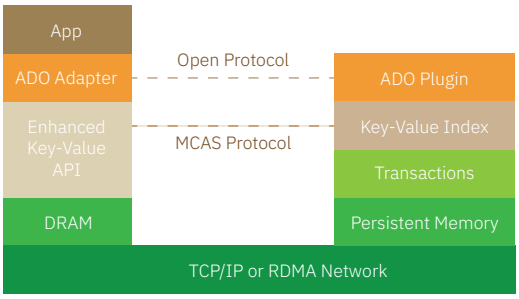


Figure 9: ADO Open Protocol Architecture

```
{
  "shards" :
  [
    {
      "core" : 0,
      "port" : 11911,
      "net"  : "mlx5_0",
      "default_backend" : "hstore",
      "dax_config" : [{
          "path": "/dev/dax0.0",
          "addr": "0x9000000000" }],
      "ado_plugins" : [
        "libcomponent-adoplugin-rustexample.so",
        "libcomponent-adoplugin-passthru.so"
      ],
      "ado_cores" : "2",
      "ado_params" :  {
        "param1" : "some param",
        "param2" : "and another"
      }
    }
  ],
  "ado_path" : "/mcas/build/dist/bin/ado",
  "net_providers" : "verbs"
}
```

Listing 2: Example MCAS two-shard configuration

Personalities can be used to create layered services and functionality, both common and domain-specific. Example common services include replication, encryption, erasure-coding, versioning, tiering, and snapshots. Domain-specific services are centered around data types, e.g., matrix operations, custom indices, and regular expression matching. Protocols are typically defined using Google Flatbuffers, but other RPC frameworks can be used.

Function	Description
`invoke_ado`	Invoke Active Data Object
`async_invoke_ado`	Asynchronous version of invoke_ado
`invoke_put_ado`	Invoke ADO with implicit put operation
`async_invoke_put_ado`	Asynchronous version of invoke_put_ado

Table 3: Advanced Client API

4.1 ADO Invocation

To support the exchange of messages from the client to the ADO, MCAS provides additional "invocation" APIs as summarized in Table 3. These APIs are directed at a key-value pair in an open pool. In addition to the key, the parameters include an opaque request, which encapsulates the protocol message:

```
status_t invoke_ado(
  const IMCAS::pool_t pool,
  const std::string& key,
  const void* request,
  const size_t request_len,
  const ado_flags_t flags,
  std::vector<ADO_response>& out_response,
  const size_t value_size = 0);
```

The `invoke_ado` invocation carries through to the ADO plugin on Ethe server side. Messages are passed from the main shard process to the ADO process via a user-level IPC (UIPC) queue (see Figure 10). UIPC is a user-level shared-memory region that is instantiated with a lock-free FIFO. Communications via UIPC do not require a system call. Before forwarding a message for an `invoke_ado` the shard thread locks the key-value pair so that the ADO has the appropriate ownership.

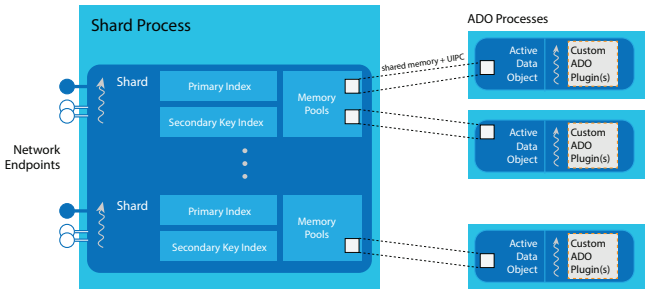


Figure 10: ADO Architecture

On receiving a message from the shard (via the UIPC queue) the ADO process calls the plugin-implemented `do_work` method. This method is defined as follows:

```
status_t do_work(
  const uint64_t              work_id,
  const char*                 key,
  const size_t                key_len,
  IADO_plugin::value_space_t& values,
  const void*                 in_work_request,
  const size_t                in_work_request_len,
  const bool                  new_root,
  response_buffer_vector_t&   response_buffers) = 0;
```

Note that the `do_work` upcall[6] provides both key-value and request information as well as an indication of whether the key-value pair has been newly created. If multiple ADO plugins are defined (see Listing 2), they are called on a round-robin schedule. Responses from the work is collected as a vector, which is ultimately passed back to the client.

On returning from the `do_work` call, the ADO container process returns a message to the shard thread via the UIPC.

[6] Moving up the stack

On receipt of the completion notification the shard thread releases the corresponding lock.

MCAS also provides the `invoke_put_ado` variation in the API. This variation allows a value to be `put` immediately prior to the invocation/upcall of the ADO plugin (avoiding the need for the client to perform two consecutive calls).

4.1.1 Experimental Signaling Hooks

There are occasions when an ADO requires notification of non-ADO invoke reads and writes (e.g., `put`/`get`) to a pool. As an experimental feature, MCAS supports the configuration of *ado signals* that relay notifications from non-ADO shard operations.

```
1    ...
2    "ado_signals" : ["post-put", "post-erase"],
3    ...
```

Listing 3: Shard-level ADO Signal Configuration

Signals are dispatched to the ADO via the UIPC queue. Before the UIPC message is sent, the corresponding key-value pair is 'read' locked. Signals propagate to the ADO plugin via a `do_work` invocation with a message prefix of 'ADO::Signal'. Responses to the clients are postponed until completion of the ADO `do_work` operation (i.e. the client is stalled).

4.2 ADO Execution Isolation

The ADO plugin manipulates data that is stored in persistent memory. The shard process exposes the subject pool memory to a separate ADO process. This exposure is based either on sharing of the fsdax file or performing a sub-space mapping with devdax. The latter requires a custom kernel module (`mcasmod.ko`) to enable mapping between processes without a file handle. Thus, the ADO process has complete visibility of the pool memory, including the primary index; it cannot read or write memory belonging to other pools.

ADO process instances effectively "sandbox" the compute performed by the developer-supplied ADO plugin. The ADO process is launched on-demand and remains operational while one or more clients has the corresponding pool open. Access to CPU cores and memory (persistent or DRAM) can be specified via the MCAS shard configuration file. Operations performed in the ADO generally cannot "jam up" or steal resources from the shard process.

Even though the ADO process has access to the primary index memory, it does not manipulate this region of memory directly (avoiding thread conflicts and the need for locking). Instead, to perform operations on the primary index, such as allocating a new key-value pair or allocating memory from the pool, a callback interface is used. The callback functions are summarized in Table 4. Invocations on the callback interface by the plugin are passed back to the shard thread via the UIPC queue.

Functions	Description
`create key/open key/erase key`	Key-value management
`resize value`	Resize existing value
`allocate/free memory`	Pool memory management
`get ref vector`	Get vector of key-value pairs
`iterate`	Iterate key-value pairs
`find key`	Scan for key through secondary index
`get pool info`	Retrieve memory utilization etc.
`unlock`	Explicitly unlock key-value pair

Table 4: ADO plugin "callback" API

As previously discussed, locking for the ADO invoke target is explicitly released when the call returns. If the ADO operation creates or opens other key-value pairs, then locks for these are also taken and added to the deferred unlock list.

4.2.1 Container-based Deployment

MCAS is cloud-native ready. Both the MCAS server process and ADO processes can be deployed as Docker containers. These containers are built with provided Dockerfiles `Dockerfile.ado` and `Dockerfile.mcas`. Furthermore, MCAS can be deployed using the Kubernates environment. `mcas-server.yaml` is provided as a reference pod configuration template.

4.3 ADO Persistent Memory Operations

Operations performed by the ADO that manipulate persistent memory must be *crash-consistent*. That is, in the event of a power-failure or reset, the data can be "recovered" to a known coherent state. For example, given a list insertion operation, in the event of recovery the list will either have the element inserted or not at all; there will be no dangling pointers or partially written data.

To support programming of crash-consistent ADO operations uses memory-transactional programming through modified standard C++ templates libraries[7]. The basic idea is to isolate the heap memory for a data structure and instrument class methods with *undo logging*. Each memory write is preceded by a copy-off operation that saves the original state of the memory that will be modified by the invocation. The copy-off operation is itself also atomically transactional. On completion of a program transaction, which is made up of one or more method invocations on the data structure, a `commit` call is made to clear the undo log. In the event of recovery from a ungraceful power-fail or reset event the undo log is checked. If the undo log is not empty, the copied-off regions are restored to the heap and then the log is cleared. This effectively "rewinds" the data structure state to before the failed

[7] We use EASTL from Electronic Arts because of its support to reset memory allocators. (https://github.com/electronicarts/EASTL)

transaction. More detail is given by the example listed in Appendix B.

Alternatively, PMDK or some other persistent programming methodology can be used to write crash-consistent ADO operations.

5 Clustering

To support coordination between MCAS processes at the ADO-level, MCAS provides basic clustering support through the *libzyre*[8] framework. This library, based on ZeroMQ[9], provides proximity based peer-to-peer networking either through UDP beaconing or a gossip protocol. It uses reliable Dealer-Router pattern for interconnections, assuring that messages are not lost unless a peer application terminates.

Clustering is enabled by adding a *cluster* section in the MCAS configuration file, such as follows:

```
{
    "cluster" :
    {
        "name" : "server-0",
        "group" : "MCAS-cluster-0",
        "addr" : "10.0.0.101",
        "port" : 11999
    },
    "shards" :
    [
        {
          ...
```

Listing 4: MCAS Process-level Cluster Configuration

Clustering is handled on a separate network port. Events, such as a node joining or leaving a cluster, are propagated to each of the shard threads and then to any active ADO processes. The plugin method `cluster_event` is up-called by the ADO thread receiving notifications via the IPC queue:

```
void cluster_event(const std::string& sender,
                   const std::string& type,
                   const std::string& message);
```

Clustering can also be deployed directly in the ADO process directly by using the cluster component (libcomponent-zyre.so). This is a useful approach when shard or pool activity needs to be associated to a different cluster group name.

6 Performance Evaluation

In this section, we highlight the performance profile of MCAS through experimental data. This results were collected from MCAS version v0.5.1 using the MPI-coordinated benchmark tool `mcas-mpi-bench`.

Our test system is based on the IBM Flashsystems platform, which provides two PCIe-sharing canisters, each with two CPU sockets. Table 5 gives the server system details. To drive the workload, four client systems are used. Even though the client systems are of similar specification, they are not precisely the same. This causes some slight perturbation of results.

[8] https://github.com/zeromq/zyre

[9] https://zeromq.org/

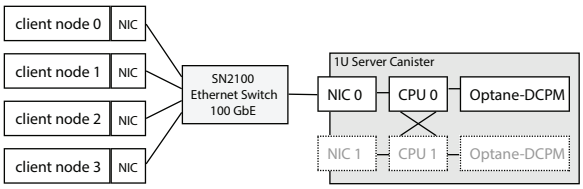


Figure 11: Experimental Network Topology

Component	Description
Processor	Intel Xeon Gold 5128 (Cascade Lake) 16-core 2.30GHz
Cache	L1 (32KiB), L2 (1MiB), L3 (22MiB)
DRAM	PC2400 DDR4 16GiB 12x DIMMs (192GiB)
NVDIMM	Optane PMM 128GB 12x DIMMs (1.5TB)
RDMA NIC	Mellanox ConnectX-5 (100GbE)
OS	Linux Fedora 27 with Kernel 4.18.19 x86_64
Compiler	GCC 7.3.1
NIC S/W	Mellanox OFED 4.5-1.0.1

Table 5: Server system specification

The server and client nodes are connected to a 100GiB Ethernet Switch (see Figure 11).

6.1 Small Operations Throughput Scaling

We examine the aggregate throughput of small `put` and `get` operations (8-byte key, 16-byte value) with increasing number of shards. 100% read (`get`) and 100% write (`put`) workloads are measured. Each shard is deployed as a separate process. Five independent client threads, with separate pools and random keys, drive the workload for each shard. All client calls are synchronous and the data is fully persistent and consistent on return.

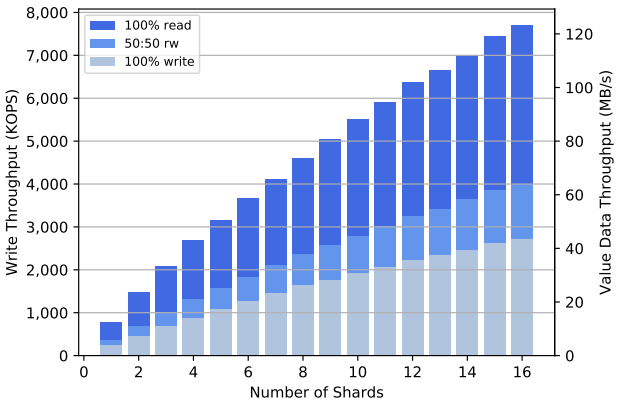


Figure 12: Scaling of Small Operations

The data shows that for 16 shards (i.e. a fully populated socket), the system can support 2.72M puts/second and 7.69M gets/second. At 16 shards, total degradation[10] is 39% and 29%, for 100% read and 100% write respectively.

6.2 4KiB Operations Throughput Scaling

We now examine throughput for `put` and `get` of larger 4KiB values. Key size remains at 8 bytes. Note, this is not using the zero-copy APIs (`put_direct` and `get_direct` that are typically used for values above 128KiB. The non-direct APIs must perform memory copies on both client and server side.

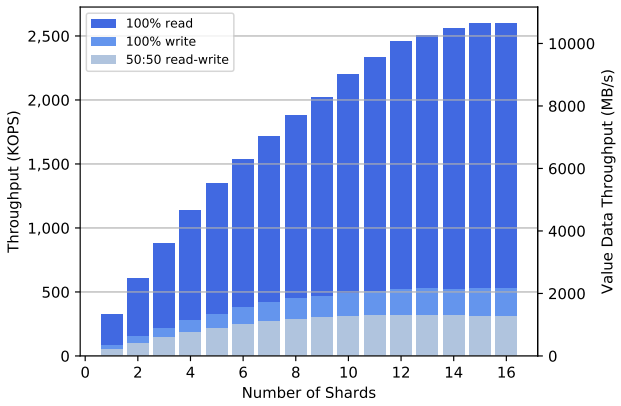


Figure 13: Scaling of 4KiB Put/Get Operations

Here read performance scales well up to around 12 shards. Write performance is considerably less.

6.3 Aggregate Scaling at Different Value Sizes

We now examine aggregate throughput for 16 shards on a single socket with network-attached clients (see Figure 11). Figure 14 shows data for 100% read and 100% write workloads. Values are doubled in size from 16B to 128KiB. Note that the operations are non-direct (i.e. copy based).

The results show that value sizes of 8KiB and above can saturate over 90% of the 100GbE bandwidth.

6.4 128KiB Direct Operations Throughput Scaling

We now examine performance scaling for the `put_direct` and `get_direct` zero-copy operations. Direct operations are not used on smaller values due to overhead of performing scatter/gather DMA.

For `get_direct`, the 100GbE RDMA network connection becomes the predominant limiter. Network bandwidth is saturated at 6 shards. For `put_direct`, the persistent memory

[10]Calculated from linear projection of observed single shard performance.

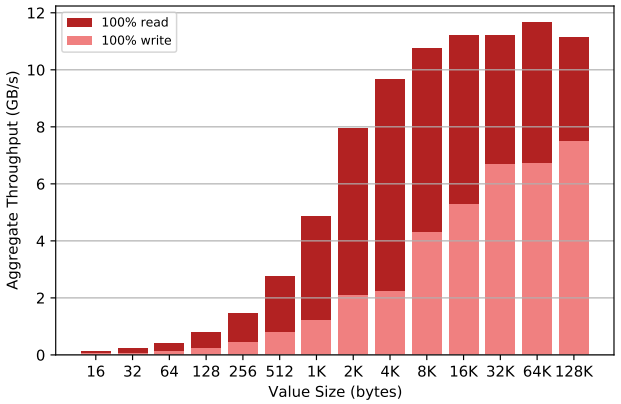


Figure 14: Aggregate-Scaling of Throughput for Changing Value Size

is the limiter (at just over 10GiB/s). This is congruent with findings reported in [7].

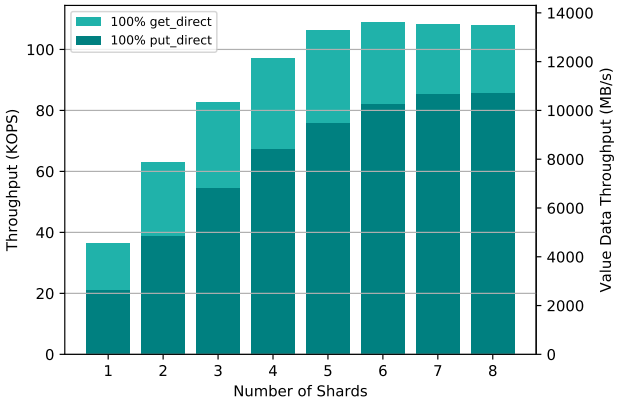
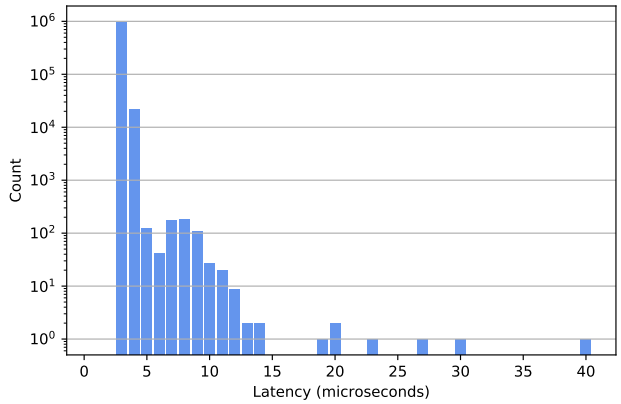


Figure 15: Shard-Scaling of 128KiB Direct Put/Get Operations

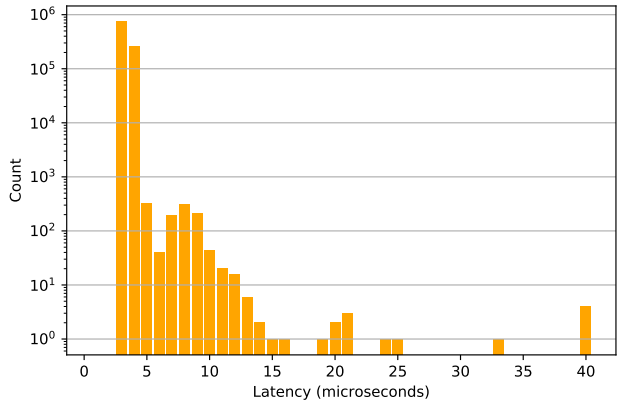
6.5 Latency

A key differentiator for Persistent Memory is the ability to achieve high throughput with low latencies. The data is provided as a histogram for 1 million samples across 40 bins. Note that the y-axes are presented with a logarithmic scale.

To provide the reader some comparison with SSD NVMe, figure 18 shows the relationship between throughput and latency for NVMe-over-fabric (same network).

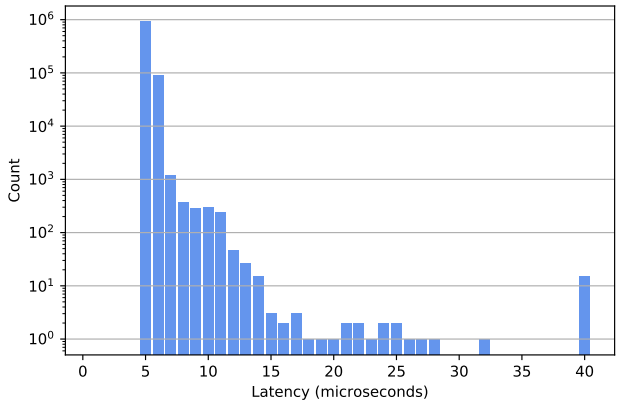


(a) Get (100% read)

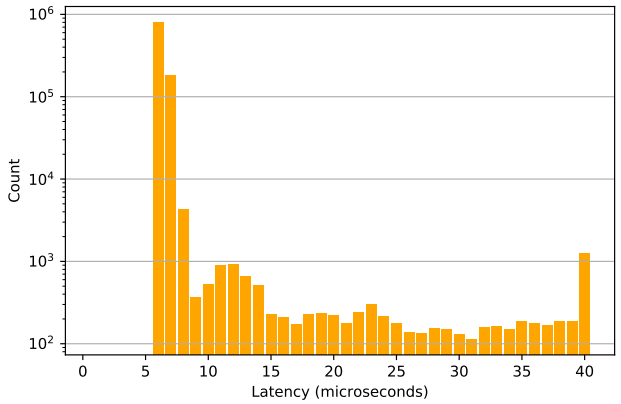


(b) Put (100% write)

Figure 16: Latency Distribution for 16B Operations



(a) Get (100% read)



(b) Put (100% write)

Figure 17: Latency Distribution for 4KiB Operations

6.6 Client Aggregation Fairness

Here we look at how client performance is affected by "competition" on the same shard. To do this, we collected throughput measures (16-byte 100% get operations) for an increasing number of threads.

Saturation occurs at as little as four client threads. Thereafter, some nominal degradation occurs, but sharing is relatively fair (i.e. each client gets its n-th share of performance where n is the number of clients).

6.7 ADO Invocation Throughput Scaling

To examine the throughput of ADO invocations in MCAS, we use the `ado_perf` tool. The test makes invocations using an 8-byte message payload. On the server, the shards are configured with the "passthru" plugin (`libcomponent-adoplugin-passthru.so`) which performs no real compute. The results shown are for a single-key target, and a pool of keys target. In the latter, 100K keys are used that belong to the same pool.

The results show that at 16 shards, aggregate throughput is 7.5M IOPS and 4.3M IOPS for same-key and key-set tests respectively. Degradation from linear is 1.4% for same-key and 18% for key-set.

7 Further Work

Currently, MCAS is a research prototype. As we mature the platform to a more robust production-grade solution, the following new features will be considered:

- Additional language bindings (e.g., Rust, Go, Java).
- Pool level authentication and admission control.
- Improvements to crash-consistent ADO development (e.g., via h/w or compiler support).
- Unified ADO protocol for enhanced key-value operations.
- Boilerplate code for client-side replication.

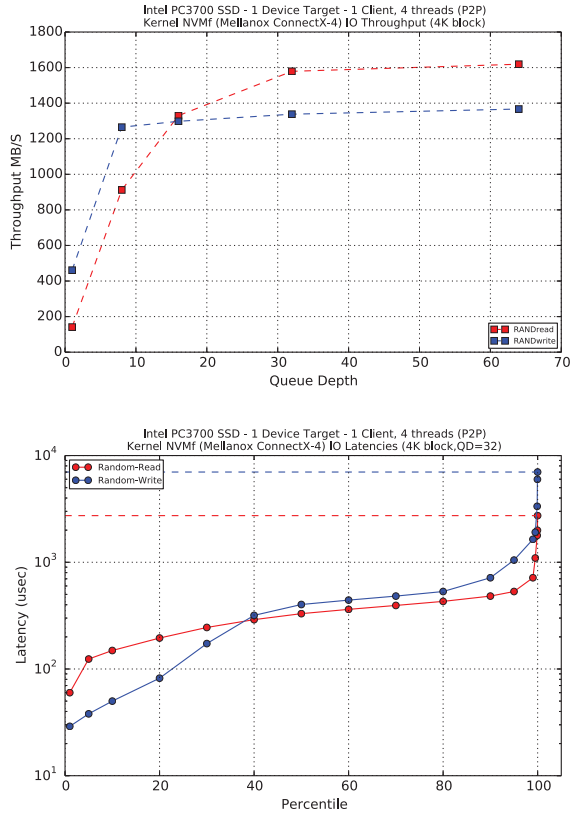


Figure 18: Throughput-Latency Relationship for NVMe-over-fabric

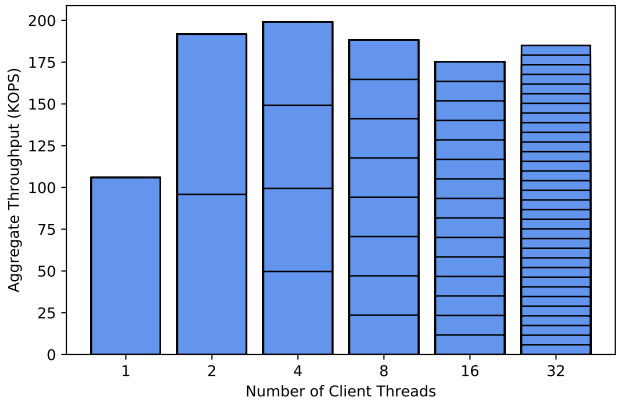


Figure 19: Aggregation Behavior

- Boilerplate code for FPGA accelerator integration.

8 Availability

The code for MCAS is available under an Apache 2.0 open source license at https://github.com/IBM/mcas/. Additional information is also available at https://ibm.github.io/mcas/.

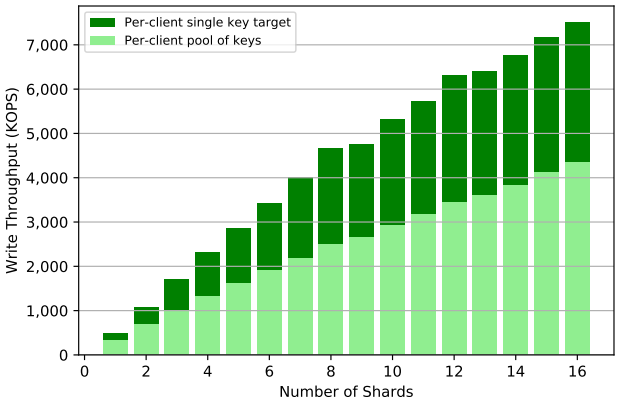


Figure 20: Scaling of ADO Invocations

9 Conclusion

MCAS is a new type of system that aims to offer memory-storage convergence. Based on persistent memory and RDMA hardware, MCAS extends the traditional key-value paradigm to support arbitrary (structured) value types with the potential for in-place operations. Our initial experiments show that the ADO approach can result in a significant reduction in network data movement and thus overall performance.

Going forward, our vision is to take MCAS beyond persistent memory and position it for emerging Near-Memory Compute (NMC) and Processing-in-Memory (PIM) technologies [3,4,6,8–10]. These technologies will be vital in addressing the memory-wall and processor scale-up problems.

References

[1] Spectra - Digital Data Outlook 2020. https://spectralogic.com/wp-content/uploads/digital_data_storage_outlook_2020.pdf, 2020.

[2] D. Bittman, D. D. E. Long, P. Alvaro, and E. L. Miller. Optimizing systems for byte-addressable NVM by reducing bit flipping. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 17–30, Boston, MA, Feb. 2019. USENIX Association.

[3] H.-M. Chen, C.-L. Hu, K.-Y. Chang, A. Küster, Y.-H. Lin, P.-S. Kuo, W.-T. Chao, B.-C. Lai, C.-N. Liu, and S.-J. Jou. On eda solutions for reconfigurable memory-centric ai edge applications. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[4] S. Gupta, M. Imani, and T. Rosing. Exploring processing in-memory for different technologies. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI

'19, page 201–206, New York, NY, USA, 2019. Association for Computing Machinery.

[5] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, pages 350–364, 2008.

[6] M. Imani, S. Gupta, Y. Kim, and T. Rosing. Floatpim: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 802–815, New York, NY, USA, 2019. Association for Computing Machinery.

[7] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[8] R. Karam, S. Paul, R. Puri, and S. Bhunia. Memory-centric reconfigurable accelerator for classification and machine learning applications. *J. Emerg. Technol. Comput. Syst.*, 13(3), May 2017.

[9] S. Khoram, Y. Zha, J. Zhang, and J. Li. Challenges and opportunities: From near-memory computing to in-memory computing. In *Proceedings of the 2017 ACM on International Symposium on Physical Design*, ISPD '17, page 43–46, New York, NY, USA, 2017. Association for Computing Machinery.

[10] A. Morad, L. Yavits, S. Kvatinsky, and R. Ginosar. Resistive gp-simd processing-in-memory. *ACM Trans. Archit. Code Optim.*, 12(4), Jan. 2016.

[11] C. Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, Apr. 2008.

[12] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, Upper Saddle River, NJ, 2 edition, 2007.

[13] D. Waddington, C. Dickey, L. Xu, T. Janssen, J. Tran, and D. Kshitij. Evaluating intel 3d-xpoint nvdimm persistent memory in the context of a key-value store. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 202–211, 2020.

Appendix

A. MCAS protocol definition

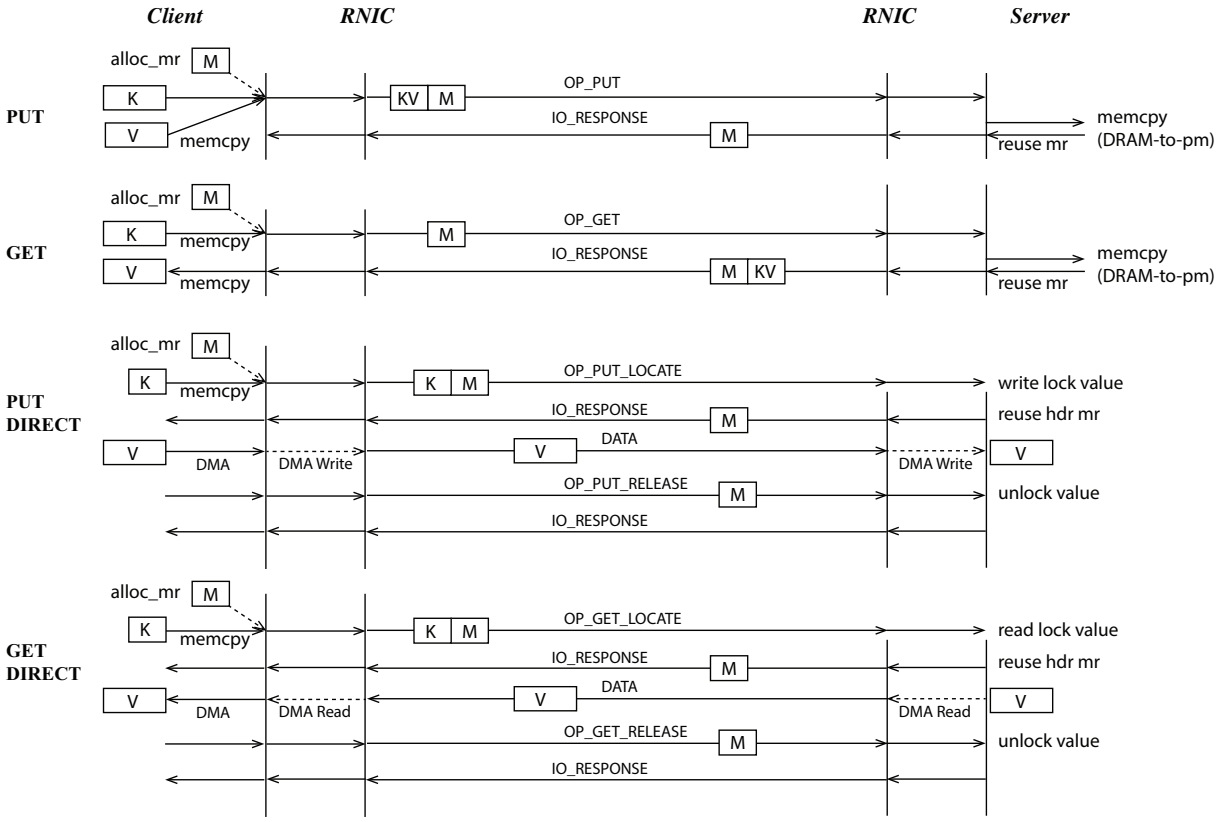


Figure 21: MCAS Protocol

B. Example C++ template-based crash-consistent programming

```
/* define crash-consistent bitset type */
using cc_bitset = ccpm::container_cc<eastl::bitset<52,
                                                   std::uint64_t,
                                                   ccpm::allocator_tl::tracker_type>>;

/* (re-)establish memory for crash-consistent heap allocator */
auto heap_regions = ccpm::region_vector_t(heap_area, heap_size);

ccpm::cca * cca; /* pointer to crash-consistent allocator */
cc_bitset * ccbitset; /* pointer to bitset data structure */


if (new_root) { /* new_root == true when key-value is first instantiated */
  /* create new allocator */
  cca = new ccpm::cca(heap_regions);
  /* create new bitset instance and pass in allocator */
  ccbitset = new (cca->allocate_root(sizeof(cc_bitset))) cc_bitset(*cca);
  ccbitset->commit();
}
else {
  /* reconstitute allocator */
  cca = new ccpm::cca(heap_regions, ccpm::accept_all);
  /* cast re-based root pointer */
  ccbitset = reinterpret_cast<cc_bitset*>(::base(cca->get_root()));
  ccbitset->rollback();
}

/* make changes to bitset data structure */
ccbitset->container->flip(4);
ccbitset->container->flip(5);
ccbitset->commit();

/* or make changes and roll back */
ccbitset->container->flip(6);
ccbitset->rollback();
```

C. Example using ADO for multi-versioning

In this section, we describe a simple ADO example that adds versioning to the basic key-value store. [11] The ADO creates an area in persistent memory that saves multiple versions of values for a specific key and allows the client to retrieve prior versions of a value. In the ADO layer we "raise" the `get` and the `put` operations into the ADO handling above the basic client API (see Table 1). The code is split into client-side library and server-side plugin. The message protocol implementation is based on flatbuffers.

A client can invoke `put` and `get` operations (see Code Listing 5). Under the hood, `put` and `get` invocations result in calls to `invoke_put_ado` and `invoke_ado` respectively.

Corresponding messages are constructed and sent as part of the ADO invoke payloads. In both operations, the target is a specific pool and key pair. The messages are transmitted from the client over the network to the main shard process and then transferred to the ADO process via user-level IPC (see Section 4.1 for more detail).

The ADO plugin handling starts with an up-call to the `do work` function (see Code Listing 6). Here, the message is unpacked and then dispatched to the appropriate `put` or `get` handler. The root pointer for the data structure that handles the different versions is provided as part of the `do work` invocation. If it is the first-ever creation of the key the ADO plugin must initialize the root data structure. In this example, the versioning metadata operations are made crash-consistent by using a basic undo log to ensure power-fail atomicity. In this example, we are using the `pmemlib` library and explicit 64-bit transactions[12].

During `put` invocation, the ADO handler creates (and persists) an undo log that records the prior value and then perform a transaction. On successful completion of the transaction, the undo log is cleared. When the system starts the ADO checks for the need to recover. If the undo log is not clear, the logged data is copied back to the original location in memory. Finally, the result of `put` and `get` operations are packed into a flatbuffer message and returned to the client.

[11] https://github.com/IBM/mcas/blob/master/examples/personalities/cpp_versioning/

[12] Building handcrafted crash-consistency can be complex but there are frameworks such as PMDK that can help to write crash-consistent code

```
status_t Client::put(const pool_t pool,
                     const std::string& key,
                     const std::string& value)
{

  /* create request  message */
  ...
  s = _mcas->invoke_put_ado(pool,
                            key,
                            fbb.GetBufferPointer(),
                            fbb.GetSize(),
                            value.data(),
                            value.length() + 1,
                            128, //root value length
                            component::IMCAS::ADO_FLAG_DETACHED,
                            response);
  return s;
}

status_t Client::get(const pool_t pool,
                     const std::string& key,
                     const int version_index,
                     std::string& out_value)
{

  /* create request  message */
  ...
  s = _mcas->invoke_ado(pool,
                        key,
                        fbb.GetBufferPointer(),
                        fbb.GetSize(),
                        0,
                        response);
  return s;
}
```

Listing 5: Client-side for ADO versioning

```
status_t ADO_example_versioning_plugin::do_work(const uint64_t work_key,
                                        const char * key,
                                        size_t key_len,
                                        IADO_plugin::value_space_t& values,
                                        const void *in_work_request,
                                        const size_t in_work_request_len,
                                        bool new_root,
                                        response_buffer_vector_t& response_buffers)
{
  auto value = values[0].ptr;
  auto root = static_cast<ADO_example_versioning_plugin_root *>(value);
  if(new_root) {
    root->init();
  }
  else {
    root->check_recovery();
  }

  if(msg->element_as_PutRequest()) {
    ...
    //  Put
    auto value_to_free = root->add_version(detached_value, detached_value_len, value_to_free_len);
  /* create response message */
    return S_OK;
  }
  else if(msg->element_as_GetRequest()) {
    ...
    // Get
    root->get_version(pr->version_index(), return_value, return_value_len, timestamp);
  }
  /* create response message */
  ...
  return S_OK;
}

void init()
{
    pmem_memset_persist(this, 0, sizeof(ADO_example_versioning_plugin_root));
}

void check_recovery() {
    /* check for undo */
    if(_undo.mid_tx()) {
       /* recover from the undo log and then clear the undo log */
      _values[_current_slot] = _undo.value;
      _undo.clear();
    }
 }


void * add_version(void * value, size_t value_len, size_t& rv_len)
{
   void * rv = _values[_current_slot];
   rv_len = _value_lengths[_current_slot];

   /* create undo log for transaction */
   _undo = { _current_slot, _values[_current_slot], _value_lengths[_current_slot], _timestamps[_current_slot] };
   pmem_persist(&_undo, sizeof(_undo));

   /* perform transaction */
   _values[_current_slot] = value;

   pmem_persist(&_current_slot, sizeof(_current_slot));

   /* reset undo log */
   _undo.clear();

   return rv; /* return value to be deleted */
}

void get_version(int version_index, void*& out_value, size_t& out_value_len, cpu_time_t& out_time_stamp) const
{
   int slot = _current_slot - 1;
   while(version_index < 0) {
     slot--;
     if(slot == -1) slot = MAX_VERSIONS - 1;
     version_index++;
   }
   out_value = _values[slot];
}
```

Listing 6: Server plugin for ADO versioning