Memory-Side Protection With a Capability Enforcement Co-Processor

LEONID AZRIEL, Technion
LUKAS HUMBEL and RETO ACHERMANN, ETH Zurich
ALEX RICHARDSON, University of Cambridge
MORITZ HOFFMANN, ETH Zurich
AVI MENDELSON, Technion
TIMOTHY ROSCOE, ETH Zurich
ROBERT N. M. WATSON, University of Cambridge
PAOLO FARABOSCHI and DEJAN MILOJICIC, HPE

Byte-addressable nonvolatile memory (NVM) blends the concepts of storage and memory and can radically improve data-centric applications, from in-memory databases to graph processing. By enabling large-capacity devices to be shared across multiple computing elements, fabric-attached NVM changes the nature of rack-scale systems and enables short-latency direct memory access while retaining data persistence properties and simplifying the software stack.

An adequate protection scheme is paramount when addressing shared and persistent memory, but mechanisms that rely on virtual memory paging suffer from the tension between performance (pushing toward large pages) and protection granularity (pushing toward small pages). To address this tension, capabilities are worth revisiting as a more powerful protection mechanism, but the long time needed to introduce new CPU features hampers the adoption of schemes that rely on instruction-set architecture support.

This article proposes the Capability Enforcement Co-Processor (CEP), a programmable memory controller that implements fine-grain protection through the capability model without requiring instruction-set support in the application CPU. CEP decouples capabilities from the application CPU instruction-set architecture, shortens time to adoption, and can rapidly evolve to embrace new persistent memory technologies, from NVDIMMs to native NVM devices, either locally connected or fabric attached in rack-scale configurations. CEP exposes an application interface based on memory handles that get internally converted to extended-pointer capabilities.

This article presents a proof of concept implementation of a distributed object store (Redis) with CEP. It also demonstrates a capability-enhanced file system (FUSE) implementation using CEP. Our proof of concept shows that CEP provides fine-grain protection while enabling direct memory access from application clients

5

L. Azriel and L. Humbel made equal contributions.

Authors' addresses: L. Azriel and A. Mendelson, Technion, Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion - Israel Institute of Technology, Haifa 32000, Israel; emails: leonida@campus.technion.ac.il, avi.mendelson@tce.technion.ac.il; L. Humbel, R. Achermann, M. Hoffmann, and T. Roscoe, ETH Zürich, Systems Group, CAB E 71.2, Universitätstrasse 6, 8092 Zürich, Switzerland; emails: {lukas.humbel, reto.achermann, moritz.hoffmann, troscoe}@inf.ethz.ch; A. Richardson and R. N. M. Watson, University of Cambridge, The Computer Laboratory, William Gates Building,15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom; emails: {Alexander.Richardson, robert.watson}@cl.cam.ac.uk; P. Faraboschi and D. Milojicic, HPE, 1501 Page Mill Road, MS1123, Palo Alto, CA 94304, USA; emails: {paolo.farabochi, dejan.milojicic}@hpe.com.

1544-3566/2019/03-ART5
https://doi.org/10.1145/3302257

to the NVM, and that by doing so opens up important performance optimization opportunities (up to 4× reduction in latency in comparison to software-based security enforcement) without compromising security.

Finally, we also sketch how a future hybrid model could improve the initial implementation by delegating some CEP functionality to a CHERI-enabled processor.

CCS Concepts: • **Security and privacy → Hardware-based security protocols**; **Database and storage security**; • **Software and its engineering → Distributed memory**; • **Computer systems organization** → *Processors and memory architectures*;

Additional Key Words and Phrases: Rack scale memory organization, software capabilities, byte-addressable persistent memory

ACM Reference format:
Leonid Azriel, Lukas Humbel, Reto Achermann, Alex Richardson, Moritz Hoffmann, Avi Mendelson, Timothy Roscoe, Robert N. M. Watson, Paolo Faraboschi, and Dejan Milojicic. 2019. Memory-Side Protection With a Capability Enforcement Co-Processor. *ACM Trans. Archit. Code Optim.* 16, 1, Article 5 (March 2019), 26 pages.
https://doi.org/10.1145/3302257

1 INTRODUCTION

Future memory systems are expected to adopt byte-addressable nonvolatile memory (NVM) devices in many arrangements. For example, NVM can be directly attached to the memory bus and treated as private memory by a single compute node, or fabric attached and shared among several compute nodes. Regardless of the specific arrangement, the nature of the "main memory" in terms of size, amount of memory that can be directly accessed by the computational subsystem, and persistence of memory itself will substantially change.

Data-centric and memory-centric architecture proposals, such as Berkeley's FireBox [4], The Machine from Hewlett Packard Enterprise [8], and Intel's Rack Scale Architecture [28], target a composable infrastructure where resources can be dynamically assigned based on application needs. Resources include massive storage, large pools of persistent memory, and thousands of computing cores. A fast memory-semantics fabric (based on protocols like RDMA or Gen-Z [20]) is the other key component to achieve the necessary coupling and scalability. This technology shift is already yielding major changes in the application domain. New scalable in-memory applications and the services they are built on, such as key-value object stores [40, 41], graph stores [48], data analytics [57] services, and transaction systems [34], are some of the examples that benefit from data-centric rack-scale architectures.

There is, however, a tension between the performance requirements of these in-memory data-intensive applications and the fine-grain protection advocated by security practices. Because translation and protection historically operate at the same granularity (a virtual memory page), a performance-driven choice of large pages (to optimize Translation Lookaside Buffer (TLB) utilization and coverage) negatively impacts security by forcing a coarse-grain protection scheme. Applications such as key-value stores (KVSs) need memory protection at a wide size range, starting from tens of bytes for keys and up to gigabytes for the values. When globally addressable memory is also a shared resource, software, including the Operating System (OS) running on the individual nodes, cannot be trusted in managing accesses to the global memory. A rack-wide entity working in cooperation with trusted components in the nodes is required to manage allocation, deallocation, and access to the shared memory pool. This approach still relies on fabric translation through standard page-based mechanisms and only offers coarse-grain protection, which hinders fine-grain access control and sharing of individual objects among nodes.

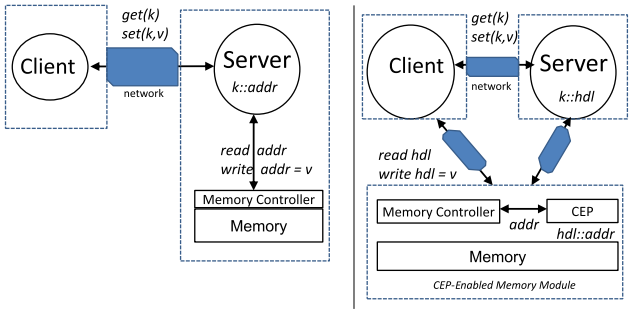

Fig. 1. CEP concept in a key-value application context.

The performance, usability, and security sweet spot tuned for the traditional block-based storage devices is suboptimal for byte-addressable rack-scale memory. Using block-access semantics inhibits many new memory-like usage models. For this reason, capabilities (originally conceived in the 1960s [39]) are worth revisiting. Capabilities are a broad concept that encompasses various mechanisms of token-based access control. In this article, we restrict it to memory-protection objects that enforce access permission to various memory zones.

Although capabilities were proven to be a useful concept, many previously proposed hardware implementations were based on the use of Instruction-Set Architecture (ISA) extensions. Unfortunately, extending the ISA of a processor is not an easy task, and the long delay between introducing the concept and the implementation of new ISA features hampers the adoption of schemes that rely on ISA support, such as CHERI [13, 62] or CODOMS [58]. It can easily take more than 5 years from conception to market introduction, delaying the impact and reducing the commercial interest.

To address these limitations, we propose a memory-side Capability Enforcement Co-Processor (CEP) that resides between the application CPU and the memory it protects (Figure 1). The left side of Figure 1 shows a traditional client-server interface implementing, for example, a simple set/get KVS interface where all memory-based operations are being executed and controlled via an intermediate call to the server that implements the *k::addr* mapping and enforces the protection. Even if the target memory is shared, clients have to go through the server to access it, because that is the only way to enforce access permissions. This process is suboptimal: data traverses the fabric at least twice, from memory to the server, and from the server to the client. Ideally, the server would return the data pointer so that the client can directly load what it needs (possibly much less than the full object), and data only traverses the fabric once. Using today's protocols, the client's access rights would extend to a full page boundary, far beyond the individual object, resulting in a fragile approach. With CEP (right side of Figure 1), the server needs to be accessed at most once to return a handle (*hdl*, through a *k:hdl* table) to the guarded data. Subsequently, the client can directly access memory by using the handle *hdl*. CEP intercepts the access before the memory controller, enforces the right access with respect to the allowed capabilities, and finally converts the handle to an actual address to perform the media access. Since the handle internally uses capabilities through CEP, we can guarantee protection. With CEP, as shown in Figure 1, we can also "free" the media from the server and attach it directly to the fabric so that accesses from multiple clients can reach the media directly without server intervention.

We believe that all components in a complex rack-scale system should be self-protecting, not having to trust the individual nodes software (including their OS), and CEP follows this design guideline. CEP implements load/store semantics that applications use to access protected memory. However, not all memory needs to be protected by CEP, and parts of the address space (e.g., the

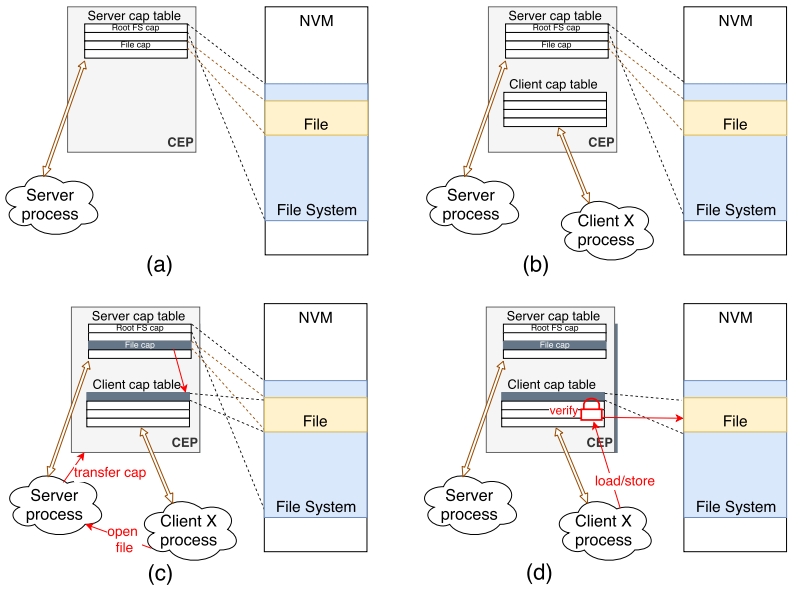

Fig. 2. CEP-based NVRAM file system example.

process stack or code) can be directly mapped to standard memory devices. The proposed architecture can still use the existing paging mechanisms and be backward compatible to all applications that do not need the use of the protected areas.

Although CEP itself implements a full capability model (in our case, we adopt CHERI), it exposes a simplified interface to the application CPU based on memory handles. This enables CEP to preserve the same level of protection as CHERI by converting memory operations to capabilities before accessing the CEP-protected memory.

In this article, we make the following key contributions:

- We propose an architecture that decouples the ISA of the application CPU from capabilities. Capabilities are implemented by the CEP instruction set, and CEP enforces the protection to the memory region it controls.
- We show how 2 important use cases (a key-value object store and a user space file system) can take advantage of the functionality that CEP offers.
- We extend the CHERI capability model by applying capabilities to physical memory, multiple address spaces, rack scale, and persistent memory, and by adding support of legacy hardware with unmodified application processors.

Figure 2 demonstrates the CEP concept showing how CEP enables a simple "in-memory" file system. In this example, a server implements a file system in NVRAM, enabling processes running on untrusted nodes to access files according to corresponding permissions. The process starts when the server has acquired a capability providing permission to access and manipulate a contiguous region in NVRAM that it will treat as a file system (Figure 2(a)). The server rights for that memory region allow it to create files with the appropriate access permissions (Figure 2(b)). Each file is assumed to consume a contiguous memory region, but in reality the server may decide to use any other (segmented) representation of the file. A similar approach is taken by many cloud-based file systems, such as Windows Azure Storage (WAS) [9] or the Google File System [21]. Files can be created, deleted, and their protection changed via a special API between the server and the client

processes. When a client process *X* wants access rights to the file, it needs to send a message to the server requesting permission to perform the operation (Figure 2(c)). The server verifies the client process *X* rights and, if approved, sends a message to CEP asking to grant access for client process *X* to the file (i.e., the memory region the files maps to, with the corresponding capability). CEP copies the capability from the server metadata table (containing the entire file system mapping data) to the local table, which persists in CEP per each client process (creating a new one if no such table exists). The reference (pointer) to the local entry within the CEP client process table is called *handle*, and CEP returns the handle to the requesting process. From then on, the client process *X* can access the data within the file directly via CEP (Figure 2(d)), through the handle, without any further access to the server.

Without CEP, the only way to gate the access is to rely on the server, or to rely on the OS of the client process node to enforce the correct access rights for the in-memory file mapping. Relying on the server creates additional work for the server, becomes a scalability bottleneck, and adds latency to the data access. With CEP, as soon as the client process gets its handle, it can communicate directly with the memory-side CEP controller, which is optimized to support such requests. A parallel memory system composed of several modules effectively includes multiple CEPs (one per memory module) and can efficiently distribute the accesses without incurring the bottleneck of a single server. A CEP-based architecture is more efficient and scalable to support massive number of parallel accesses to the NVRAM system and can scale to a large number of nodes (and processes) connected to the fabric. Note that in the whole process, CEP's only function is resolution of capabilities and mapping from process-specific handles to capabilities. All the intelligence about memory management is in the application server hands.

In the absence of a real hardware CEP, the implementation we use in this article is a software-emulated proof of concept. An optimized CEP hardware design would significantly improve the effectiveness of our system. We address possible hardware optimizations later in the article.

2 BACKGROUND

In today's OSs, a process is associated with a single Virtual Address Space (VAS), and access to physical memory is protected by the memory management unit (MMU) hardware. The OS programs the MMU and the hardware caches (TLBs), but—except for handling page faults—does not intercept memory accesses from user space.

Paged virtual memory was introduced when physical memory was a scarce resource, giving applications the illusion of more memory than available. With an increasing amount of physical memory, demand paging rarely happens. However, first-level TLBs have remained relatively small (64 4KB-page entries [27]), and while larger page sizes mitigate the TLB pressure, they artificially increase the protection granularity and still do not provide a complete solution [6].

Emerging persistent memory technologies are expected to enable a load/store interface rather than block based. Yet OSs mostly assume a block-based interface [22] to persistence and struggle to achieve the potential performance benefits offered by fast NVM devices to legacy applications [14, 16]. As NVM closes the performance gap with DRAM, OS-enforced policies become impractical because of the prohibitive overhead of intercepting individual loads and stores. Support for safe, low-overhead, fine-grain, user-level mechanisms to access and manage NVM regions becomes mandatory for memory-centric programming models.

The introduction of NVM poses another problem: persistent data survives the lifetime of a process, and something must ensure the validity of persistent pointers. Because processes can only dereference pointers to virtual addresses, storing pointer-rich data structures in persistent memory can lead to dangling or invalid references to volatile resources when regions are remapped at another address. This also implies that protection at the virtual level is not sufficient.

Finally, many security breaches rely on memory corruption, such as buffer or stack overflows, to gain unauthorized access to a system. Memory bugs often occur because of the lack of fine-grain protection: page-level hardware protection cannot prevent unauthorized access within a page. Rack-scale systems aggravate this problem because of the load/store interface to globally accessible shared memory.

2.1 Capability Systems

The 1960s and 1970s witnessed the first appearance of approaches to address authorization with capability-based hardware and software techniques.

Hardware-supported capability systems like CAP [47], StarOS [32], and IBM System/38 [24] extend the ISA with special instructions and registers allowing hardware to enforce protection for even small objects without mediation of a trusted entity. Other systems proposing hardware capabilities include the M-Machine [12] (adopting a capability system without compatibility requirements), CHERI [62] (retaining compatibility with capability-unaware code), and CODOMs [58] (focusing on hardware support for isolation between components).

Kernel-supported capabilities do not rely on hardware support: only the kernel has the privilege to perform capability operations. OSs such as Chorus [51] and Amoeba [45] rely on sparsity and cryptography to make capabilities unforgeable. In Hydra [15, 38, 63], KeyKOS [49], EROS [55], Mach [1, 19, 26, 53], and Accent [50], capabilities are kernel objects, applications only have handles to these objects, and they are only directly accessible by the kernel. Systems like Barrelfish [7] and L4 [23] also adopt this technique. Kernel capabilities either provide protection at page level or require kernel invocation for using smaller objects that discourage the use of kernel capabilities. In distributed capability systems (e.g., Barrelfish), the system maintains a capability derivation tree to provide distributed capability revocation so that only a set of safe operations to modify capabilities can be executed. Similar approaches have been followed by L4Re [37] and seL4 [35] to mediate memory and object access through kernel-protected capabilities.

Finally, another way of implementing capabilities is at the language level. Programming languages such as E [43], Joe-E [42], and Caja [44] rely on compiler and runtime to enforce a strict object-capability model. Although lower-level languages like C have a hard time to achieve this strong type safety, several approaches exist to prevent out-of-bound accesses to data structures. Software-based solutions include Softbound [46], Cyclone [31], and low-fat pointers (software variant) [18]. Hardware-supported bound checks aim at reducing the software overhead and include techniques like Hardbound [17] and low-fat pointers [36].

2.2 CHERI

CHERI implements a hardware capability model by extending the 64-bit MIPS instruction set. The new CHERI instructions provide byte-granular memory protection features and support for compartmentalization. Similar to previous work on Capsicum [60], CHERI uses a hybrid capability model, extending rather than replacing existing security mechanisms. For example, CHERI still allows page-based protection and supports existing code.

Replacing pointers with capabilities provides many advantages to users. For example, hardware bounds checking prevents buffer overflows, and the use of permissions ensures that access restrictions are observed (e.g., read-only data cannot be written to). When capabilities are stored in main memory, overwriting them with plain data never results in legally dereferenceable pointers, thus eliminating attacks such as return-oriented programming [54]. On a higher level, CHERI also provides an object capability model (i.e., capabilities that refer to software objects) for compartmentalization.

CHERI uses 256 bits to represent capabilities: 64 bits each for base, length, and offset of the capability, and 64 bits for permissions and type fields. A 128-bit compressed representation also exists [61]. Tagged memory (memory that contains a tag bit near every memory location that indicates whether this location stores a capability) and explicit management instructions make capabilities unforgeable. An additional protected bit in the physical address space indicates whether a memory location holds a valid capability. Any modifications of that memory location from a non-CHERI instruction clears the capability bit and transforms it into regular data (which cannot be used as a regular pointer to reference memory). CHERI instructions enforce that capability rights can only be restricted and never increased.

From an application perspective, CHERI capabilities provide fine-grain access control to volatile objects, operate on virtual addresses, and are consequently tied to a single VAS. This approach has limitations when multiple virtual addresses can refer to the same shared object in a single node, or shared across a rack-scale system. Because hardware support for CHERI is at the ISA level, CHERI supports capabilities within the VAS. This has advantages and challenges. Advantage is in robustness; when the VAS is deleted, all capabilities are revoked. The challenge is that all support required for persistence and sharing across address spaces is very limited and nonexistent at the moment because it requires capabilities support on the physical address space [62].

Although CHERI capabilities provide full *spatial* memory safety by preventing out-of-bounds accesses, they do not provide *temporal* memory safety. It is not feasible to find and efficiently remove all derived capabilities after a `free()` operation. This operation is called *revocation* and becomes crucial once we start sharing capabilities among untrusted processes and as we move to persistent memory with load/store semantics. An incorrect revocation enables processes to access data using stale capabilities, which is an open invitation to exploit the vulnerability for malicious purposes.

Our approach borrows many of the basic concepts of CHERI but implements them without the need for ISA extensions. Adopting this approach has the benefit of flexibility in the implementation. It can be seen as a bridge until the final ISA support appears in CPUs, or as an alternative when ISA support is not desired or sufficient. Furthermore, thanks to the choice of independent processor-based implementation, our approach provides greater flexibility in adding features, such as the revocation support.

2.3 Other Related Work

An alternative to capability hardware is through the OS, but that comes with significant performance overhead because each capability operation requires crossing the kernel boundary. We were motivated by the way the OS maintains derivation trees in support of revocation, and, in a way, we can see our approach as an accelerator for OS-supported capabilities.

Other hardware-based protection approaches, such as Intel's SGX [3] and MPX [29], and ARM's TrustZone [2], serve different purposes in support of different applications. For example, MPX supports a limited form of bounds checking in conjunction with compilers. SGX and TrustZone provide trusted environments for safe execution of code. CEP can take advantage of functionality such as MPX to implement fast client-side checking of capabilities and reducing some of the CEP transactions.

The way in which CEP operates shares some similarities to RDMA, which is very efficient in moving large data chunks (using a put/get interface). Both rely on a memory-side entity that mediates accesses to memory between a client (issuing the memory access) and a server (hosting the memory being accessed). RDMA was also being used for implementing file systems [30] and KVS [33]; however, RDMA requires a memory registration step to pin remote memory regions, and the protection mechanism relies on region keys that, unlike capabilities, are guessable and forgeable.

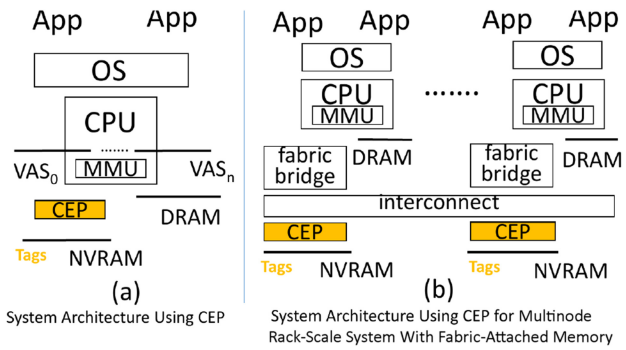

Fig. 3. CEP for single (a) and multiple (b) CPU nodes.

3 ARCHITECTURE AND DESIGN

Our objective in designing CEP is to implement a hardware-enforced capability system where applications can run on an off-the-shelf unmodified general-purpose CPU. To protect memory accesses, we introduce the CEP co-processor(s) in the path between the application CPU and protected memory. The CPU directly manages memory that does not need the extra protection, without CEP intervention. CEP serves as an intermediary between CPUs and protected memory, which may or may not be nonvolatile (NVM) or shared among multiple computing elements.

Because the CEP protection check happens on the memory side, this approach works for single nodes (when memory is private) and at rack scale (when memory is shared (Figure 3)). Although we focus on NVM as the most likely technology candidate of fabric-attached memory, CEP also covers DRAM, or any other form of directly addressable memory, without any loss of generality. In a single-node system, CEP is a co-processor that offloads CHERI capability operations from the CPU (Figure 3(a)). One can think of CHERI as a "smart memory controller" with a rich interface and capable of executing more complex operations than simple reads and writes. At rack scale (Figure 3(b)), CEP is a memory-side controller, not dedicated to a specific CPU node but acting as a shared capability co-processor for all the nodes. This is conceptually similar to the role that the NVMe over Fabric (NVMeoF) controller plays today for fabric-attached block-based devices. At the physical layer, the CPU communicates with CEP via the existing memory channel, either directly attached or over a memory fabric. For load balancing and increasing memory bandwidth, the system can include multiple CEPs associated with multiple memory modules and channels. CEP operates with physical (or fabric) addresses, after the virtual memory system (hardware MMU) has completed the node-local translations and protection checks.

In a CEP-enabled system, the part of memory protected by CEP is not accessible using the regular load/store instructions from the CPU. Instead, application software uses a dedicated per-process communication channel over the memory fabric network, where the fabric identifiers couple the application process to its corresponding CEP context. Specific implementations of the coupling mechanism depend on the fabric and require structures such as the communication buffer interfaces in Heterogeneous System Architecture (HSA) [25].

CEP provides Quality of Service to the multiple clients it serves by a fair arbitration mechanism. The arbitration policy and implementation may vary depending on the system configuration and application objectives. In our prototype implementation, the CEP software launches a separate thread for every process it communicates with (see Section 6). The OS kernel running on CEP then facilitates the fairness. If the client OSs are not trusted, additional protection is needed, for example, to prevent a Denial of Service (DoS) attack by a malicious node flooding CEP with fake requests.

A system composed of multiple nodes can contain multiple CEPs, each of which controls a specific memory region or memory channel. From this perspective, one can think of CEP as an active component in the media controllers in charge of accessing memory and enforcing protection.

Since we assume that an application CPU lacks native support for capabilities, the capabilities to access protected memory have to be contained in CEP and cannot be visible outside of the CEP "trusted" domain. User processes and the OS running on the application CPU can only refer to capabilities through "handles." To distinguish between processes, each CEP provides a dedicated communication channel to every process in every application CPU client. Namely, a CEP communication channel uniquely identifies a client process. The application OS is trusted to assign the channels correctly to user processes. A compromised OS can obtain rights of all of its processes by impersonating them, but the memory fabric node identification mechanism prevents it from obtaining rights of another node.

The *(node_id, pid, handle)* tuple unambiguously identifies a capability. Handles are meaningful only within a process context. Since handles are process specific, processes are our unit of trust. Multiple threads of the same process have shared context in CEP. Synchronization and other shared data management is done at the node, whereas the memory fabric guarantees order. Unlike CHERI, CEP does not offer any functionality for address space compartmentalization. Because of this locality constraint, handles cannot be directly passed between processes. To delegate capability to another process, a process must invoke a *transfer* operation on CEP. In POSIX, a new process is created by calling *fork* from a parent process. File handles are similar to capability handles, and those stay valid in the child process. Hence, a good fit for the existing semantics is to clone the whole capability table in CEP and set it as the lookup table of the newly created process. This maintains the validity of all handles, but each process has its own handle space. We did not investigate the cost of this clone operation, but since the expensive part is updating the derivation graph, we expect it to behave similarly to a series of capability transfers. Hence, an implementation should, due to the cost, make the capability space clone optional.

CEP's implementation uses a CHERI-enabled simple microcontroller embedded in CEP itself. To convert from handles to capabilities, by default, CEP software maintains a fixed-size translation table for each process. To support varying table size requirements (e.g., an application server process may need a bigger table than a client), a table of a specific size may be requested from CEP during the process bootstrap. Although capabilities are persistent, handles are ephemeral objects, meaningful only within the process context and therefore disappearing when the process dies. Although using fixed-size tables speeds up access time, it complicates the implementation, as tables can run out of space. Various mechanisms can be used to address this, such as having pools of tables with different size (e.g., memory pools [56]), allowing to expand tables, or allowing the process to delete existing entries to make space for capabilities when the table is full (to be used with caution because of the side effects). We leave the full support for extensible tables as future work.

To derive a more restricted (in size or rights) capability from an existing one, a user process provides a handle to CEP, which searches for an unused handle in the translation table, stores the restricted capability, and returns the allocated handle to the user process. The derivation rules follow the CHERI principles of monotonic nonincrease of rights. An alternative way for the process to obtain a new capability is to *transfer* between processes. As described earlier, a process is identified by the combination of a unique node id, provided by the memory fabric with guaranteed integrity, and a process id within the node assigned by the node's OS (or hypervisor). Absent a node id integrity protection mechanism, CEP can also use a central authority to guarantee unique process identifiers. The details of the global allocation scheme are beyond the scope of this article.

The sequence to transfer a capability works as follows. Assume that process $P1$ wants to transfer a capability to process $P2$. $P1$ sends a transfer request to CEP indicating the identifier of $P2$. CEP

copies a delegated capability to $P2$'s capability table and returns a handle (valid in the $P2$'s context) to $P1$. $P1$ then sends $P2$ this handle.

Note that CEP does not provide an option to create a new capability from scratch. To preserve a consistent protection model, capabilities can be derived only from other capabilities. Therefore, bootstrapping the system requires an external entity (e.g., the global fabric manager) to generate an initial "root" capability that enables access to the entire fabric memory space. The description of this "once in a lifetime" procedure is outside the scope of this article, but it is similar to how other capability-based systems operate.

CEP enforces memory protection only when a process attempts dereferencing the protected memory through a handle. It exposes a simple interface (see Section 3.1) by which an application can efficiently request protected memory accesses.

3.1 The CEP Programming Interface

A client process wanting to access the protected memory must invoke the CEP API. The CEP application-side library transforms the API calls into requests placed on the CEP's process-specific bidirectional queue. In the folllowing, we list the most common operations of the CEP API (note that each operation returns a status/error code, which we omit for brevity):

- void CStore(cap_handle_t handle, int32 offset, bytes[] data)
  Store data at *handle* at *offset*. The data is copied to the command queue by the library.
- void CStoreC(cap_handle_t handle, int32 offset, cap_handle_t cap)
  Store capability *cap* in the capability referred to by *handle* at *offset*.
- bytes[] CLoad(cap_handle_t handle, int32 offset, int32 data_size)
  Load from *handle* at *offset data_size* bytes. The response contains an error code and, in case of success, the read data.
- cap_handle_t CLoadC(cap_handle_t handle, int32 offset)
  Load capability from region pointed to by *handle* at offset *offset* bytes. Returns a handle to the loaded capability.
- proc_id_t CGetIdentity()
  Return process identifier of the calling process.
- cap_handle_t CTransfer(cap_handle_t handle, proc_id_t dest_id)
  Transfer to capability associated with *handle* to process *dest_id*. Returns a handle valid for client.
- void CRevoke(cap_handle_t handle)
  Revoke a capability. After the function returns, CEP guarantees that there are no more in-flight accesses to derived capabilities, and all future access produce a revoked error.
- void CInvalidate(cap_handle_t handle)
  Clear (not revoke) a capability.
- cap_handle_t CDerive(cap_handle_t src, int32 offset, size_t size, cap_perm_t perm)
  Derive capability from *src* with *size* at *offset* and permissions *perm*. An error is returned if the range or permissions exceed those of *src*.
- struct metadata CGetMetaData(cap_handle_t handle)
  Return metadata, such as size, permissions, and revocation status.

Handles are appended with node and process identification, whereas the handle itself is translated to a table index. The node and process identifiers are used to determine the correct communication buffer, and the client of the API is not aware of the handle format. Once a handle is valid, CEP never invalidates it autonomously. In the case of revocation, CEP marks it as revoked. A revocation error is returned on the next load or store operation. The client should then explicitly

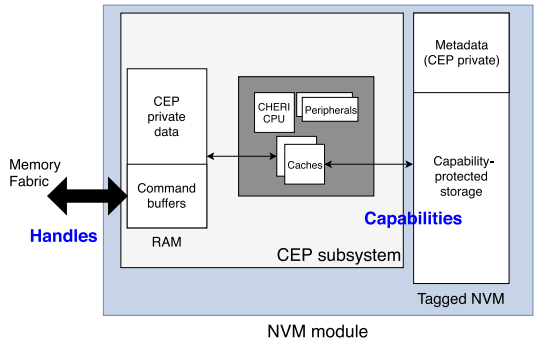

Fig. 4. Architecture of the CEP co-processor.

invalidate the handle and remove all client-side references to it. Note that invalidation (clearing) the capability removes the entry indexed by the handle in the process capability table and frees the space for new entries. In contrast, revocation is a recursive operation that marks all the child capabilities as revoked. See Section 3.3 for details.

The command API described earlier uses a synchronous, blocking protocol, where every call blocks until acknowledged by CEP. We assume that the fabric supports arbitrary-size memory semantics operations, similar to what protocols such as RDMA or Gen-Z implement.

Load and store operations provide an offset and length, which can be used by applications to avoid overfetching unnecessary data. In case of a store operation, the library implementing these functions copies data to the command buffer. In case of a load operation, the library returns a pointer into the command buffer containing the retrieved data. The client is expected to consume the data before issuing the next request.

To facilitate programming, we have also implemented a C++ template wrapper consisting of 2 classes: cap_ptr<T> and cap_val<T>. cap_ptr<T> encapsulates a cap_handle_t and offers an object-oriented interface to the capability operations. cap_val<T> is a proxy object that forwards assignments to CStore and forwards casts to T to CLoad. The dereference operation turns a cap_ptr<T> into a cap_val<T>.

3.2 CEP System Architecture

We designed CEP with an embedded microcontroller that supports the full set of CHERI capabilities embedded in its ISA. Since the CEP firmware is trusted, we could also use an unmodified ISA and a full software implementation. We chose a capability-enabled ISA because of performance and designed it so that it is more robust against critical security bugs. In addition, a processor-based CEP opens opportunities for offloading computation securely to a near-memory programmable controller. These offloaded plugins may run in a secure compartment with lower access rights by leveraging the CHERI compartmentalized execution environment support [62].

In addition to ISA capability support, CHERI comprises a tagged memory mechanism. In tagged memory, an additional bit is attached to each memory location that indicates whether this memory location stores a capability or just regular data. Modifications to this bit are implicitly done on capability operations in memory, such as writing a capability.

We envision an NVM-based system as a promising CEP target and assume a globally addressable NVM divided into multiple *NVM modules*. Each NVM module maps a contiguous segment of memory and is managed by a controller containing a dedicated CEP. CEP can issue byte granular loads and stores to the corresponding NVM segment (Figure 4). Each CEP subsystem also includes volatile RAM as standard working memory. The RAM holds ephemeral CEP private data, such as

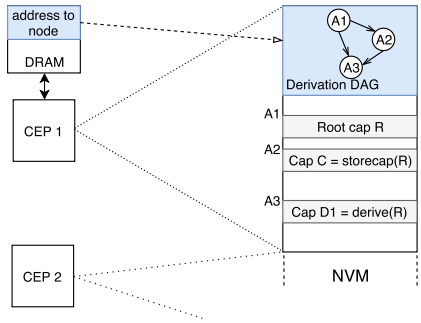

Fig. 5. Derivation graph.

code and stack. Furthermore, in the memory-mapped communication model planned for future versions, it also holds the communication buffers with the CPUs (see Section 7). All state that is needed to restart a system is stored in NVM.

CEP acts as a gateway between the CPU nodes and NVM, and mediates every access to the protected segment it controls. The resolution of the specific CEP-to-target mapping is part of the memory fabric protocol. For simplicity, we assume a statically partitioned global fabric address space (i.e., a flat fabric space where an address uniquely identifies each NVM module in the system).

CEP runs a simple OS (or microkernel) with a single main process acting as a server for clients running within the nodes. The server runs a dedicated thread (servlet) for every client, which maintains circular request and response queues in the dedicated memory-mapped communication buffers. A thread pool approach could be deployed to limit the number of concurrent connections and throttle client traffic in case of congested resources.

3.3 Revocation Support

One of the significant challenges in protecting shared resources in a multinode environment is how to revoke capabilities. With persistent memory, revocation becomes an even more critical functionality. Unlike the case of volatile memories, process restarts and reboots do not free up NVM. The introduction of memory-side CEP, which acts as the ultimate access controller for any access on the way to protected memory, is a fundamental enabler to implement a robust revocation scheme.

To allow revocation, CEP maintains a derivation-directed acyclic graph (DAG) that tracks links between capabilities. Any operation creating a new derived capability is recorded in that graph. The derivation DAG is a global structure in the CEP context residing entirely in the CEP-protected metadata region only accessible by CEP itself (Figure 5). Every capability in the CEP space has an associated node in the derivation DAG. The nodes contain links to the parents and to the children. The nodes reference capabilities by their physical (or CEP virtual) addresses, but there are no backward references from capabilities to their requesting clients. This allows us to use a standard CHERI capability format within CEP. To provide the backward links, CEP maintains a RAM-based hash table that maps physical addresses of capabilities to derivation DAG nodes. In addition to improving access time, the RAM-based hash table potentially reduces NVM wear-out by removing write traffic. CEP reconstructs the hash table at power-up and keeps it consistent with the NVM structure during normal operation.

The derivation DAG provides the following methods:

- *addNode*: Create a root node.
- *addLinkedNode*: Create a node connected to a parent.

- *copyNode*: Create a new node with replicated links from the source node.
- *invalidateNode*: Invalidate a capability and remove the corresponding node connecting all the node's parents to all of its children.
- *revoke*: Invalidate the node and recursively all its successors, unless a successor has more than 1 parent.

A capability is invalidated by setting its size to 0. In addition, a user-defined permission bit is cleared to indicate that this capability has been revoked. CEP can later use this flag to send the client a revoked error upon dereferencing a revoked capability. The *addLinkedNode* method is invoked to derive and transfer capabilities between CEP servlets. Indirection (loading and storing capabilities via other capabilities) invokes the *copyNode* method.

With multiple CEPs, each CEP is in charge of its own NVM segment. Thus, a CEP manages its own derivation DAG that corresponds to the capabilities that point to the segment's address space. Derivation links can only live within a single context, thus no cross-CEP links are ever required.

4 SECURITY AND TRUST MODEL

CEP inherits some of the security properties of the underlying CHERI protection model, particularly the monotonic nonincrease of rights and tagged memory. Unlike CHERI, CEP mediates access to the protected memory and disables direct CPU access to the protected memory region. CEP associates each capability to a unique handle that it passes to the clients, and it exclusively owns the handle-to-capability translation table and all the metadata associated with the capability-based storage.

With CEP, every process running on the application CPU uses a dedicated bidirectional communication channel, which uniquely identifies the process. CEP uses this information to authenticate requests, so a memory handle is only valid within the corresponding process context. We rely on the OS to maintain the integrity of communication channels of the processes running on that OS. A leaked handle, due to programming error, for example, is invalid outside of its designated process, as it can only be used in the correct communication channel to CEP.

CEP is part of the Trusted Computing Base (TCB) of a system. System hardware, including the processor and its peripherals, the memory subsystem, and the interface to protected memory, is also included in the TCB. CEP software must also be trusted, similarly to any other system firmware component. We assume that it cannot be modified by an untrusted component and that updates are secure (e.g., signed).

Another component of the TCB is the interconnect fabric. We assume that the fabric protocol ensures the integrity of data and authenticity of components. The fabric also provides unambiguous identification of a node during communication with CEP and a secure communication channels among CEP instances. These features are common, and modern protocols, such as Gen-Z [20], include security extensions that cryptographically provide these guarantees.

We also rely on the application OS to enforce separation between process contexts so that a process can only access the assigned CEP communication channel (which is how CEP identifies the process). A compromised OS only affects the handles it hosts. To mitigate against OS compromises, we can delegate the management of communication channels to a trusted execution environment, such as ARM's TrustZone, Intel's SGX [3], or a hypervisor. This would, however, cause a higher overhead for secure context switches. Regardless of the approach adopted, the impact of a compromised OS is always bound by the rights granted to the corresponding node, which CEP can control. In contrast, in a CEP-less system that uses page-based sharing, a compromised OS is able to access *all* memory.

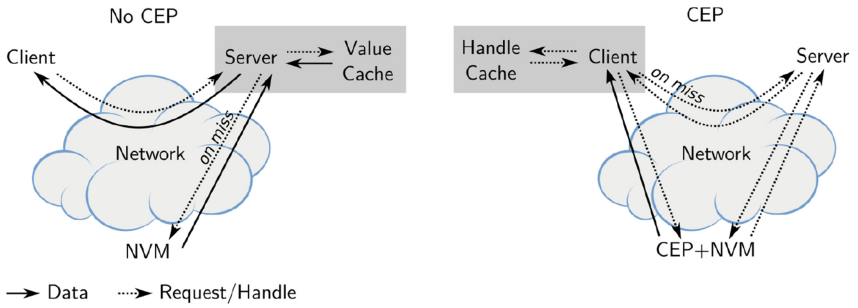

Fig. 6. Standard object store dataflow (left) and how CEP reduces the network traffic (right).

At rack scale, a trusted management server is responsible for the allocation (and deallocation) of fabric-attached memory access rights by securely programming the CEP to return the correct capabilities to application requests from a given CPU.

5 USE CASES

This section describes 2 use cases that we ported and evaluated to a CEP-enhanced (simulated) system: a key-value object store (Redis) and a capability-enhanced file system (FUSE).

5.1 Object Store

Key-value object stores have become one of the most important foundational services for distributed applications, as they provide support for application persistence at scalability, cost, robustness, and performance that surpasses traditional databases.

We selected Redis [52] as our initial target, but the mechanisms we describe also apply to other frameworks where a server transfers in-memory data to a client. The Redis interface offers a rich set of operations; here, we focus on *set* and *get* because they are used most frequently.

Our baseline is a standard Redis implementation in a distributed rack-scale setting with fabric-attached memory. The client and server are separate programs residing on different compute nodes (and OSs) but sharing access to a common memory pool through a network that supports direct memory operations. The client issues a *get* request by sending it across the network to the server; the server determines the data memory location, fetches the data from memory over the network, and returns it to the client, again over the network (left side of Figure 6). In this scenario, data traverses the network at least twice. Since all the value accesses are mediated by the server, the server can choose to cache values in its local memory. The client, however, cannot cache values because the client is unaware of data changes that other clients may initiate and would not know when to invalidate the cache.

CEP reduces the data traffic by transferring references to data instead of data by value. The server returns a capability handle to memory, and the client uses that handle to access the memory to retrieve the value. This model also allows client-side caching of capability handles to further improve performance and reduce server load.

We modified the Redis server such that every value is referenced by a separate capability. On the initial server start, the server issues a requests to the memory manager and obtains a guarded (capability protected) memory area for storing its value data. On server restarts, the server reloads the persisted structure from a capability-enabled file system. In both cases, we obtain a capability c_{all} that allows the server to access all guarded data allocated to the KVS data structures.

The server maintains a memory allocator structure to manage the guarded storage. To store a value for a new key, the server uses the allocator to find space in the guarded area c_{all} and uses CEP

to derive a smaller capability $c_s \subset c_{all}$ pointing to the newly allocated area. The server then stores the value to be associated with the key in c_s and updates its internal key to a capability handle lookup structure to associate the key with c_s. When a client requests to *get* a key, the server does not fetch the data but merely transfers the capability to the client avoiding to fetch the value from memory in the first place.

The client receives a capability handle that the client can use through CEP to fetch the data directly from memory, reducing the number of data transfers over the network. As in the NoCEP setting, it is impossible for the client to maintain a key to value cache because of the absence of a coherency protocol. However, the client can safely maintain a key-to-capability cache. This is because CEP checks on every access if the capability is valid. The server can perform cache invalidations by revoking the capability. When a capability is revoked, the client receives a notification on the next access. The client can then fall back to querying the server. The server typically responds with a new capability, but it can also inform the client of other conditions, such as when the key was deleted.

The client can also use the capability to update data values if the c_s contains enough space to hold the new value. To ensure atomicity, clients must use memory synchronization primitives. If the data is larger than the allocated space, the client must involve the server requesting a reallocation of a larger buffer from its memory pool. The server can then update its data structures and revoke the old capability. When this happens, other clients with a cached capability will receive a (lazy) revocation error on the next access, which they can use in a recovery process that falls back to ask the server for a new capability. Section 6 provides a detailed breakdown of the operations involved in this process.

5.2 Capability-Enhanced File System

In addition to object stores, file systems are another universal persistence service that provides a useful abstraction over raw storage blocks. Typically, an application has no direct access to the storage blocks but must use OS-assisted DMA to transfer data to a main memory buffer. Access control is usually at the granularity of directories and files, whose ACL metadata stores permissions for the owner, specific groups, or others.

Our proposed capability-based approach addresses these inefficiencies. With NVM as storage, processors could directly access the contents of the durable medium with a byte-addressable load/store memory interface. Because CEP validates each access, it can also enforce protection at byte granularity using capabilities.

This, in turn, leads to a powerful combination of files and capabilities: while the standard UNIX file permissions grant access to the entire file or directory, a process can distribute a capability to parts of the file to others. For instance, a key-value object store like Redis can use the file system and take advantage of known abstractions but then hand over a capability referring to a single object to one of its clients, without giving access to the entire store.

Furthermore, a capability-based file system can also facilitate bootstrapping the distribution of capabilities by providing a UNIX socket–like way to find and obtain capabilities communication endpoints.

Last, using capabilities to represent files and directories leads to a more scalable system, as path resolution can start relative to a particular *root* capability and clients can traverse the subtree locally.

As a proof of concept, we implemented a capability-based FUSE file system on Linux. Our implementation supports the legacy POSIX interface—open, read, write, close—and we extended it through a set of IOCTL commands to manage the capability for the file, or the capabilities stored within the file.

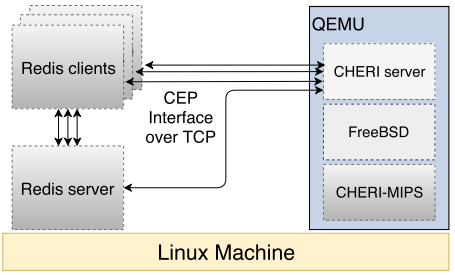

Fig. 7. Prototype system running Redis.

6 EVALUATION

In this section, we present our evaluation platform and methodology, the microbenchmarks we developed to estimate the overhead introduced by CEP, and how we combine the estimate to assess the performance of the full key-value object store application performance.

6.1 Evaluation Platform

A priority when choosing the evaluation platform was to get accurate cycle numbers for a processor that supports hardware-enforced capabilities. Hence, we use the CHERI-enhanced MIPS processor (built by the Cambridge Computer Laboratory [61]) as a platform for the CEP co-processor. We emulate a single node and single CEP system on a Linux host, where QEMU models CEP as a CHERI-MIPS processor [11] and the clients run as host processes. Within the emulated environment, we run a FreeBSD OS modified to support CHERI [10]. The CEP server runs as a single process on top of the FreeBSD OS. For each new client connection, the server spawns a new dedicated servlet thread. We emulate the NVM backend by mapping it to an area of the CEP process memory.

The application side of the system is represented by client processes running on the host Linux platform. The application processes communicate with CEP via a library that emulates the memory-mapped communication interface. The natural target for CEP is memory fabric protocols with load/store semantics (e.g., Gen-Z, CCIX, or OpenCAPI). This enables direct fine-grain access to (remote) data as opposed to going through a server. For the sake of experiment, we rely on the existing infrastructure, and hence the prototype system implements the communication on top of TCP/IP sockets. Figure 7 depicts the prototype system running a Redis application. The current implementation is synchronous, and client calls block until the server responds. Server responses contain status/error codes and, in case of a load, also the data payload.

6.2 Estimating CEP Latency Overhead

To evaluate the latency introduced by CEP, we collect traces from the QEMU emulation of the CEP co-processor while running Redis-based workloads. The traces include executed instructions and memory accesses. For the CEP implementation, we assume a single-issue MIPS, in-order microcontroller running at 1.5GHz. Keeping in mind that CEP is integrated in a memory-side controller, we believe these parameters are aligned with the cost and energy envelope available in a typical media controller of a fabric-attached NVM module. To estimate the CEP latency, we also assume that we have a single memory-side CEP targeting byte-addressable NVM, that the CEP operations are blocking (except for derivation DAG updates), and that CEP is always available to the clients. In addition, we obtained an average cycles per instruction (CPI) number from running CEP workloads on the FPGA-based CHERI development platform (BERI) [59]. The number we got can be

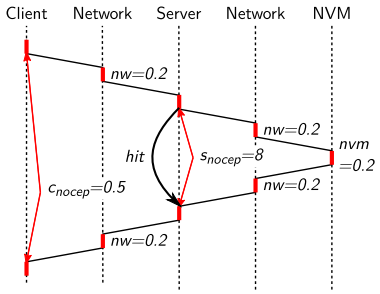

Fig. 8. Redis GET request without CEP, hit and miss (latencies expressed in microseconds).

approximated to CPI = 1.5. We take into account that the NVM is cacheable within CEP when calculating NVM-related latency. The cache hit rate is obtained from a stand-alone cache simulator that gets the QEMU traces as an input. Thus, the CEP induced latency is calculated as

$$t_{CEP} = InstCount \cdot f_{MIPS}^{-1} \cdot CPI + t_{NVMacc} \cdot MissRate.$$

6.3 Redis Performance Evaluation

To evaluate the performance of a CEP-enabled system, we combine results of application-level benchmarks (in which the time spent in CEP and network are excluded from calculations of network latency), NVM access time, and CEP overhead. The CEP overhead is calculated using instruction traces and takes caching effects into account.

We start with evaluating our baseline system (NoCEP) as described in Section 5. The evaluation makes optimistic assumptions about NoCEP. First, we exclude the network costs in our benchmark and replace them with our estimates. These are assuming a zero-copy network, whereas Redis typically communicates using the TCP/IP stack incurring an additional copy. Second, for measuring server latencies, we store a 1-byte value. This reduces the lookup time of the Redis server to a minimum. The stored data is so small that Redis holds it inside the hash table, removing the need for memory allocation. We assume that the hash table is stored in local DRAM. We also allow the Redis server to cache data in local DRAM, avoiding NVM trips for a limited amount of small-size values.

However, we are conservative in determining the CEP-enabled Redis performance. We choose a very simple algorithm for fetching a capability, and we assume a fairly slow CEP microcontroller to estimate the CEP runtime overhead. Our system offers the ability to only partially access the returned data, but we do not make use of that and transfer all the stored data. We do allow our system to cache handles on the client side, adding a hash table that resolves keys to capability handles. As we only populate the system with one key, the client caching deals with a working set equal to the one of the Redis servers. In practice, the client may only cache the keys it is using, which represent a small subset of the whole key space. The hit rate of the client cache is an important factor for performance, which we investigate. As we do for the baseline, we assume that the key lookup hash table is stored in local DRAM.

In the sequence diagram (Figure 8), we show how an unmodified Redis client operates. First, we account for some local pre- and postprocessing time c_{nocep} to set up the request and parse the response. Then we add the network latency nw and the time the Redis server takes to process the request s_{nocep} (which includes parsing of the request and looking up the storage location of the requested key). The server fetches data, again requiring a network latency nw plus the NVM access latency nvm. The CEP-enabled system depicted in Figure 9 incurs most of the NoCEP latencies, but each time we go to the NVM, we incur the CEP overhead cep_{ld}. Because the client can

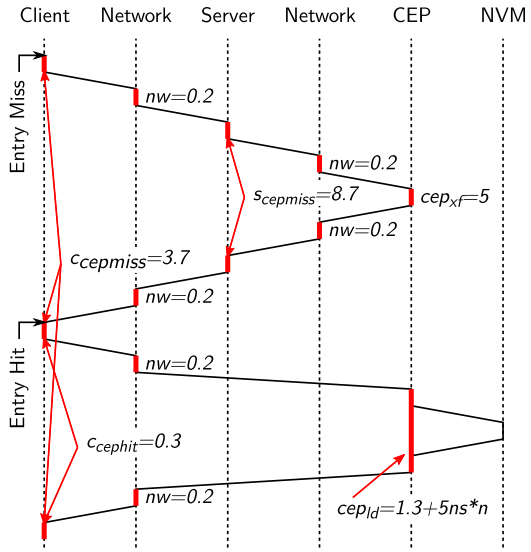

Fig. 9. Redis GET request with CEP, hit and miss (latencies expressed in microseconds).

cache requests, processing cost in the client is $c_{cepmiss}$ in the case of a cache miss or c_{cephit} on a cache hit. Furthermore, the server instructs the CEP to transfer capabilities (not data), which uses cep_{xf} processing time on the CEP. The values are listed in Table 1.

In both cases, we collected measurements on a 6-core Intel Xeon X5670 CPU with 2.93GHz and 24GiB DRAM. Each benchmark performs 100,000 iterations of the Redis benchmark.

We assemble the measurements, estimates, and simulations according to the flow depicted in Figure 8. We assume keys, memory addresses, and handles to be small constant k bytes. Now we compare the latencies of requesting n bytes in a Redis-enabled CEP versus an unmodified Redis (NoCEP).

$$
\begin{aligned}
NoCEP_{hit}(n) &= c_{nocep}(n) + nw(k) + nw(n) + s_{nocep}(n) \\
NoCEP_{miss}(n) &= NoCEP_{hit}(n) + nw(k) + nw(n) + nvm(n) \\
CEP_{hit}(n) &= 4nw(n) + c_{cephit}(n) \\
CEP_{miss}(n) &= 4nw(k) + s_{cepmiss}(n) + cep_{xf} + CEP_{hit}(n)
\end{aligned}
$$

Figure 10 shows the latencies for different payload sizes. For small requests less than 1,024 bytes, CEP provides a speedup for cache hits. With increasing payload size, the CEP cache hits become slower than in the NoCEP scenario. CEP cache misses are slower for all payload sizes, and the slowdown slightly increases with growing payloads.

If we use CEP and we hit the cache, payloads are available faster than without CEP because the client can go straight to memory. In the extreme scenario of a single-byte payload, we reduce the latency by 4.4X. With these small payloads, the CEP overhead is negligible and the processing time in the Redis server is reduced to zero. With growing payload sizes, the transfer overhead of the CEP becomes more dominant.

In the case of a miss, the CEP-enabled system is slower because of the additional key lookups on both the server and the client, and we also have to include the CEP capability transfer and CEP data load overhead.

In the case of a hit, the amount of data transferred is always smaller in the CEP scenario, whereas in the case of a miss, it depends on the payload size. Once the payload is larger than the 4 extra

Table 1. Latencies and Bandwidth for n Byte Payload

Formula	What	Latency	Legend
$c_{nocep\{hit,miss\}}$	Client, NoCEP	$0.5\mu s^{\dagger}$	† Measured, 1 byte payload, independent of n
c_{cephit}	Client, CEP, hit	$0.3\mu s^{\dagger}$	‡ Measured, 1 byte value, grows with n
$c_{cepmiss}$	Client, CEP, miss	$3.7\mu s^{\dagger}$	♯ Estimate
s_{nocep}	Server	$8\mu s^{\ddagger}$	♮ Simulation based on QEMU traces
$s_{cepmiss}$	Server, CEP, miss	$8.7\mu s^{\dagger}$	
nw	Network	$200\text{ns} + 1\text{ns} \cdot n^{\sharp}$	
nvm	NVM	$200\text{ns} + 1\text{ns} \cdot n^{\sharp}$	
cep_{ld}	CEP, Load	$1.3\mu s + 5\text{ns} \cdot n^{\natural}$	
cep_{st}	CEP, Store	$0.9\mu s + 4\text{ns} \cdot n^{\natural}$	
cep_{xf}	CEP, Transfer	$5\mu s^{\natural}$	

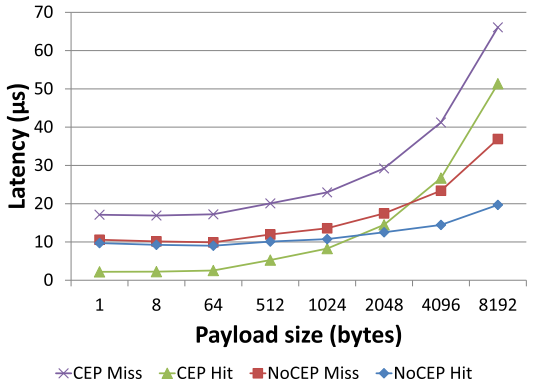

Fig. 10. Latency of increasing payload sizes assuming all hit or all miss in CEP and NoCEP Redis.

network requests carrying k bytes each, less data is transferred since the client fetches the payload directly from memory.

With our conservative CEP performance estimates, CEP cannot quite keep up with the NVM throughput for large payloads. The benefit of fewer round trips is overshadowed by the longer transfer time when the payload is big enough. CEP performance shines for small payloads, which remains an important case and difficult to optimize.

The evaluation so far has shown that cache hits become faster while cache misses become slower. Hence, we investigate which cache hit rate is necessary for the CEP system to outperform the unmodified system. Figure 11 shows the answer to this question for 2 sample payload sizes. The CEP system performs better starting with a hit rate of 49% for a 1-byte payload and with a hit rate of 62% for a 512-byte payload. Atikoglu et al. [5] have shown that in a KVS used as a caching layer at Facebook, the hit rate can reach up to 98.2%. Furthermore, their analysis shows that the value sizes in their workloads follow a power law distribution with more than 80% of the value smaller than 512 bytes. However, the clients cache may be smaller (and disjoint), which may increase compulsory misses. Yet considering the large margin between 98.2% and our 49% break-even estimate, we still expect a substantial speedup.

CEP introduces some overhead, but it allows new types of sharing, which can lead to an overall system speedup (Section 7). In our evaluated system, after a certain hit rate in the client-side caching, we see an improvement in overall performance. The exact ratio of cache hits necessary to

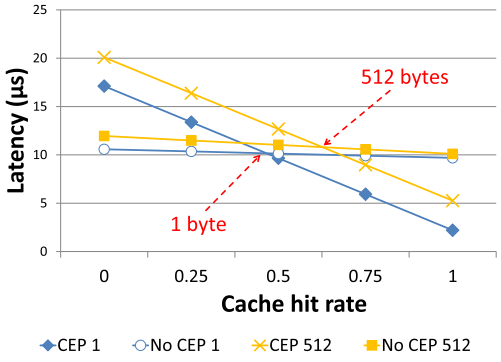

Fig. 11. Effect of cache hit rate on latency.

see an improvement depends on many factors, but even under the conservative assumptions we make, we see significant performance potential.

We also identified several additional optimization opportunities for CEP design, some of which we highlight here:

- **Improve CEP transfer performance.** The major bottleneck is fetching large objects from NVM. Our simulations use a software copy in CEP. Specialized memory-copy hardware support that takes the CPU out of the data path can significantly increase the CEP bandwidth.
- **Improve the algorithm.** When the client fetches a handle from the server, the server could, along with the capability, send the requested data similar to the NoCEP implementation. This reduces the effect of the network latency.
- **Improve CEP capability transfer performance.** Several software optimization opportunities exist to improve the CEP performance for capability operations. For capability transfer, the majority of time, about 4μs of 5μs, is spent in updating the derivation tree, which might execute asynchronously reducing the latency to about 1μs.
- **Improve the data structure.** When a client needs to access a group of keys, it could request the CEP server to return a capability pointing to other capabilities. This corresponds to returning a "view" (or projection, in the database meaning of the work) of the stored data, whose layout can be optimized for a specific access pattern. Similarly to a database, the CEP server may internally maintain multiple views for each access class and access pattern.
- **Distributing CEP.** Although we currently assume a memory-side CEP only, we could bring some of the CEPs' functionality closer to the CPU clients. This approach increases complexity, because the CEPs become a distributed system, but creates opportunities to perform some operations locally and asynchronously, which hides the network latency.

7 EVOLUTION OF THE CEP VISION

In this article, we presented a proof of concept implementation of CEP, but we believe that the potential usage space of the CEP concept is much wider. This work discusses potential additional optimizations to performance and functionality of the CEP-based system. In particular, we address systems and memory architectures evolving from directly attached volatile memory to fabric-attached rack-scale persistent memory. So far, we mainly focused on capabilities as a promising approach to solve several of the protection challenges this evolution introduces. For example, CEP mitigates the significant adoption time lag that prevents practical deployment of ISA-supported capabilities. We already discussed the benefits of using CEP to implement persistent memory and rack-scale sharing; however, our vision goes beyond this.

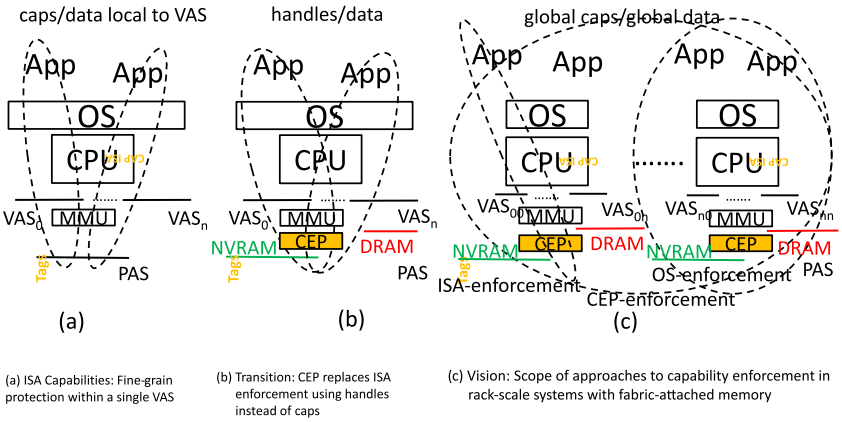

Fig. 12. Evolution of the CEP vision.

Figure 12 depicts how our vision evolved (a) from a capability system implementing ISA-supported fine-grain memory protection within virtual address space, (b) through a co-processor approach supporting shared and persistent memory, and (c) finally to a combined cooperative model where ISA capabilities can protect local volatile memory (DRAM) and CEP capabilities can protect cross-node sharing in fabric-attached and persistent memory organizations.

At some point in the future, we expect mainstream CPUs to start including some form of ISA support for capabilities, such as CHERI. In this scenario, the programming model needs to evolve toward a "hybrid" capability scheme. The node CPU can protect the local DRAM using CHERI-extended capability pointers. However, NVM access (both local and rack scale) requires a separate CEP-based scheme, because CHERI does not support persistent shared capabilities that can outlive the VAS where they were created.

A hybrid model also helps to reduce the communication cost and to increase parallelism by introducing a memory-mapped CEP communication interface. Part of CEP's local memory will be dedicated to per-process communication buffers, implemented as circular request queues that CEP continuously polls for incoming requests. The application side will implement a synchronous communication protocol, where the application software waits for a new entry in the response queue after placing the request. Performance-critical operations, such as loads and stores, bypass the queue and will be routed directly to the protected memory by a hardware bridge. The bridge will automatically enforce memory protection based on the originating request owner and the capabilities it possesses.

At rack scale, shared NVM will be accessible (given the relevant permissions) from every CPU node using load/store semantics via a coherent memory fabric. This allows for NVM data to be cached within the nodes. Although all the basic capability operations including creation, enforcement, and dereferencing will be natively supported by the CPU, the cross-node revocation scheme will remain a centralized task that requires a memory-side CEP.

The memory-side CEP co-processor snoops capability management operations, such as capability loads and stores, and maintains the derivation graph. When a capability is revoked, CEP walks the derivation graph to invalidate all the derived capabilities as described in Section 3.3. To support revocation, CEP can take advantage of memory coherency protocol between CEP and the CPU nodes. However, coherency is not strictly required since other invalidation-based software schemes could achieve similar, possibly less optimized, functionality. We leave this study for future work.

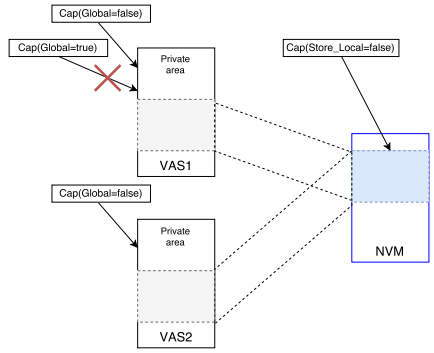

Fig. 13. NVM sharing and local capabilities.

One of the significant contributions of the proposed architecture to the traditional CHERI is support for sharing capabilities between nodes, and between different virtual address spaces. This is enabled by a combination of a forced mapping of the NVM area to the specific process's VAS and CHERI local capability support. First of all, NVM is divided into 2 sections, the *NVM metadata area*, used to store metadata, mainly the derivation tree, and the *NVM data area* accessible by the user. To enable sharing, the NVM data area is identically mapped in every virtual address. Thus, processes running on the same or on different nodes may share capabilities by storing them in a shared NVM region. The remaining memory can be mapped freely in the VAS by the OS.

Shared memory regions are vulnerable to malicious clients injecting a forged capability pointing to the victim's local space. To prevent these attacks, we propose to use CHERI's global/local capability mechanism. Capabilities with a *Global* flag unset are defined as local and can only be stored via capabilities that have the *Store_Local_Capability* set. The CHERI monotonic concept that prevents increasing rights assures that global capabilities can be converted to local, but not the reverse. Capabilities pointing to a process's private space have the *Global* bit unset, and capabilities that refer to shared NVM have the *Store_Local_Capability* flag unset. This way, no capability to a process private space can be stored in NVM (Figure 13).

In addition to the local/global capability mechanism, we rely on the OS to program the MMU so that it can enforce the local/global domain separation. If the OS cannot be trusted, hardware or other trusted entities (either a hypervisor or Trusted Execution Environment) can enforce the mapping of NVM through the MMU.

8 SUMMARY AND FUTURE WORK

CEP enables fine-grain memory protection for rack-scale NVM-based systems. We demonstrated its use on an object store and file system, foundational services of all distributed and enterprise applications, which we ported to the CEP API in a very short time and straightforward manner.

Our evaluation shows that the advantages of removing intermediary software agents compensates the CEP-introduced performance overhead for NVM accesses, resulting in overall performance improvement.

CEP provides abundant opportunities for future research. We proposed a possible solution to combine ISA-supported VAS capabilities with a near-memory protection scheme enforced by a hardware CEP. There are design alternatives to our proposal with different tradeoffs A particular route discussed uses capabilities as an additional address translation mechanism removing the need to map all NVM in each VAS. This includes enabling position-independent data to be supported by CEP and enables additional flexibility in accessing large NVM pools.

Similarly, new designs of capability-based kernels are desirable for optimal rack-scale lazy revocation. New applications designed from scratch could leverage CEP in new ways. CEP could enable safe near-memory processing, as it is aware of the access rights of each process and may support offloading of complex operations, without compromising security.

REFERENCES

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. 1986. *Mach: A New Kernel Foundation for UNIX Development*. Technical Report. Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

[2] T. Alves. 2004. Trustzone: Integrated Hardware and Software Security. White paper.

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, Vol. 13.

[4] Krste Asanovic. 2014. FireBox: A hardware building block for 2020 warehouse-scale computers. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. https://www.usenix.org/conference/fast14/technical-sessions/presentation/keynote.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Performance Evaluation Review* 40, 1, 53–64.

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. 237–248. DOI: https://doi.org/10.1145/2485922.2485943

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, et al. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. 29–44. DOI: https://doi.org/10.1145/1629575.1629579

[8] K. M. Bresniker, S. Singhal, and R. S. Williams. 2015. Adapting to thrive in a new economy of memory abundance. *Computer* 48, 12 (Dec. 2015), 44–53. DOI: https://doi.org/10.1109/MC.2015.368

[9] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, et al. 2011. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, NY, 143–157. DOI: https://doi.org/10.1145/2043556.2043571

[10] Cambridge Computer Laboratory. 2017. CheriBSD. Retrieved February 2, 2019 from https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html.

[11] Cambridge Computer Laboratory. 2017. Qemu-CHERI. Retrieved February 2, 2019 from https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-qemu.html.

[12] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware support for fast capability-based addressing. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*. ACM, New York, NY, 319–327. DOI: https://doi.org/10.1145/195473.195579

[13] David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, et al. 2015. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 117–130. DOI: https://doi.org/10.1145/2694344.2694367

[14] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, et al. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGPLAN Notices* 46, 3 (March 2011), 105–118. DOI: https://doi.org/10.1145/1961296.1950380

[15] Ellis Cohen and David Jefferson. 1975. Protection in the Hydra operating system. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP'75)*. ACM, New York, NY, 141–160. DOI: https://doi.org/10.1145/800213.806532

[16] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, et al. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. 133–146. DOI: https://doi.org/10.1145/1629575.1629589

[17] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural support for spatial safety of the C programming language. *ACM SIGARCH Computer Architecture News* 36, 1 (March 2008), 103–114.

[18] Gregory J. Duck and Roland H. C. Yap. 2016. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC'16)*. ACM, New York, NY, 132–142. DOI: https://doi.org/10.1145/2892208.2892212

[19] Alessandro Forin, Ro Forin, Joseph Barrera, Michael Young, and Richard Rashid. 1988. Design, implementation, and performance evaluation of a distributed shared memory server for mach. In *Proceedings of the 1988 Winter USENIX Conference*.

[20] Gen-Z Consortium. 2017. The Gen-Z Consortium Website. Retrieved February 2, 2019 from http://www.genzconsortium.org.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 29–43. DOI: https://doi.org/10.1145/945445.945450

[22] The Open Group. 2013. *POSIX.1-200: The Open Group Base Specifications Issue 7*. The Open Group. Available at http://pubs.opengroup.org/onlinepubs/9699919799/.

[23] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. 1997. The performance of μ-kernel-based Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, New York, NY, 66–77. DOI: https://doi.org/10.1145/268998.266660

[24] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. 1981. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA'81)*. IEEE, Los Alamitos, CA, 341–348. http://dl.acm.org/citation.cfm?id=800052.801885

[25] W. W. Hwu. 2015. *Heterogeneous System Architecture: A New Compute Platform Infrastructure*. Elsevier Science. https://books.google.co.il/books?id=otXUBQAAQBAJ.

[26] Joseph Barrera III. 1991. A fast mach network IPC implementation. In *Proceedings of the USENIX MACH Symposium*. 1–11.

[27] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Order Number: 248966-033. Available at http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[28] Intel Corporation. 2016. Intel Rack Scale Design. Retrieved February 2, 2019 from https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-architecture/intel-rack-scale-architecture-resources.html.

[29] Intel PLC. 2013. Introduction to Intel Memory Protection Extensions. Retrieved February 2, 2019 from http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[30] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, et al. 2012. High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'12)*. IEEE, Los Alamitos, CA, Article 35, 35 pages. http://dl.acm.org/citation.cfm?id=2388996.2389044

[31] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the USENIX Annual Technical Conference (ATEC'02)*. 275–288. http://dl.acm.org/citation.cfm?id=647057.713871

[32] Anita K. Jones, Robert J. Chansler Jr, Ivor Durham, Karsten Schwans, and Steven R. Vegdahl. 1979. StarOS, a multiprocessor operating system for the support of task forces. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP'79)*. ACM, New York, NY, 117–127. DOI: https://doi.org/10.1145/800215.806579

[33] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. *ACM SIGCOMM Computer Communication Review* 44, 4 (Aug. 2014), 295–306. DOI: https://doi.org/10.1145/2740070.2626299

[34] Hideaki Kimura. 2015. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM, New York, NY, 691–706. DOI: https://doi.org/10.1145/2723372.2746480

[35] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 207–220. DOI: https://doi.org/10.1145/1629575.1629596

[36] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight Jr, and Andre DeHon. 2013. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*. ACM, New York, NY, 721–732. DOI: https://doi.org/10.1145/2508859.2516713

[37] Adam Lackorzynski and Alexander Warg. 2009. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the 2nd Workshop on Isolation and Integration in Embedded Systems, Eurosys Affiliated Workshop (IIES'09)*. ACM, New York, NY, 25–30. DOI: https://doi.org/10.1145/1519130.1519135

[38] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. 1975. Policy/mechanism separation in Hydra. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP'75)*. ACM, New York, NY, 132–140. DOI: https://doi.org/10.1145/800213.806531

[39] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA.

[40] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, et al. 2016. Full-stack architecting to achieve a billion-requests-per-second throughput on a single key-value store server platform. *ACM Transactions on Computer Systems* 34, 2 (April 2016), Article 5, 30 pages. DOI: https://doi.org/10.1145/2897393

[41] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. 429–444. http://dl.acm.org.ezproxy-pa2.labs.hpe.com/citation.cfm?id=2616448.2616488

[42] Adrian Mettler and David Wagner. 2010. Class properties for security review in an object-capability subset of Java: (short paper). In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'10)*. ACM, New York, NY, Article 7, 7 pages. DOI: https://doi.org/10.1145/1814217.1814224

[43] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University, Baltimore, MD.

[44] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Caja: Safe active content in Sanitized JavaScript. Google, Inc., Tech. Rep.

[45] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. 1990. Amoeba, a distributed operating system for the 1990s. *Computer* 33, 5 (May 1990), 44–53.

[46] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, New York, NY, 245–258. DOI: https://doi.org/10.1145/1542476.1542504

[47] R. M. Needham and R. D. H. Walker. 1977. The Cambridge CAP computer and its protection system. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles (SOSP'77)*. ACM, New York, NY, 1–10. DOI: https://doi.org/10.1145/800214.806541

[48] Marcus Paradies, Michael Rudolf, Christof Bornhövd, and Wolfgang Lehner. 2014. GRATIN: Accelerating graph traversals in main-memory column stores. In *Proceedings of the Workshop on Graph Data Management Experiences and Systems (GRADES'14)*. ACM, New York, NY, Article 9, 6 pages. DOI: https://doi.org/10.1145/2621934.2621941

[49] S. A. Rajunas, N. Hardy, A. C. Bomberger, W. S. Frantz, and C. R. Landau. 1986. Security in KeyKOS. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*.

[50] Richard F. Rashid and George G. Robertson. 1981. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP'81)*. ACM, New York, NY, 64–75. DOI: https://doi.org/10.1145/800216.806593

[51] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, et al. 1991. Overview of the CHORUS distributed operating systems. *Computing Systems* 1, 39–69.

[52] Salvatore Sanfilippo. 2015. Redis. Retrieved February 2, 2019 from http://redis.io/.

[53] E. J. Sebes. 1991. Overview of the architecture of Distributed Trusted Mach. In *Proceedings of the USENIX Mach Symposium*. 20–22.

[54] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14thACM Conference on Computer and Communications Security (CCS'07)*. ACM, New York, NY, 552–61.

[55] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*. 170–185. DOI: https://doi.org/10.1145/319151.319163

[56] Michael Stonebraker. 1981. Operating system support for database management. *Communications of the ACM* 24, 7 (July 1981), 412–418. DOI: https://doi.org/10.1145/358699.358703

[57] Christian Tinnefeld, Donald Kossmann, Martin Grund, Joos-Hendrik Boese, Frank Renkes, Vishal Sikka, et al. 2013. Elastic online analytical processing on RAMCloud. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT'13)*. ACM, New York, NY, 454–464. DOI: https://doi.org/10.1145/2452376.2452429

[58] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting software with code-centric memory domains. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE, Los Alamitos, CA, 469–480. http://dl.acm.org.ezproxy-pa2.labs.hpe.com/citation.cfm?id=2665671.2665741

[59] Robert N. M. Watson, Jonathan Woodruff, David Chisnall, Brooks Davis, Wojciech Koszek, A. Theodore Markettos, et al. 2015. *Bluespec Extensible RISC Implementation: BERI Hardware Reference*. Technical Report. Computer Laboratory, University of Cambridge.

[60] Robert N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. 2010. Capsicum: Practical capabilities for Unix. In *Proceedings of the 19th USENIX Security Symposium*.

[61] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, et al. 2016. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 5)*. Technical

Report UCAM-CL-TR-891. Computer Laboratory, University of Cambridge. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-891.pdf.

[62] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, et al. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP'15)*. IEEE, Los Alamitos, CA, 20–37. DOI: https://doi.org/10.1109/SP.2015.9

[63] W. A. Wulf, R. Levin, and S. P. Harbison. 1981. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, New York, NY.

Received June 2018; revised November 2018; accepted December 2018