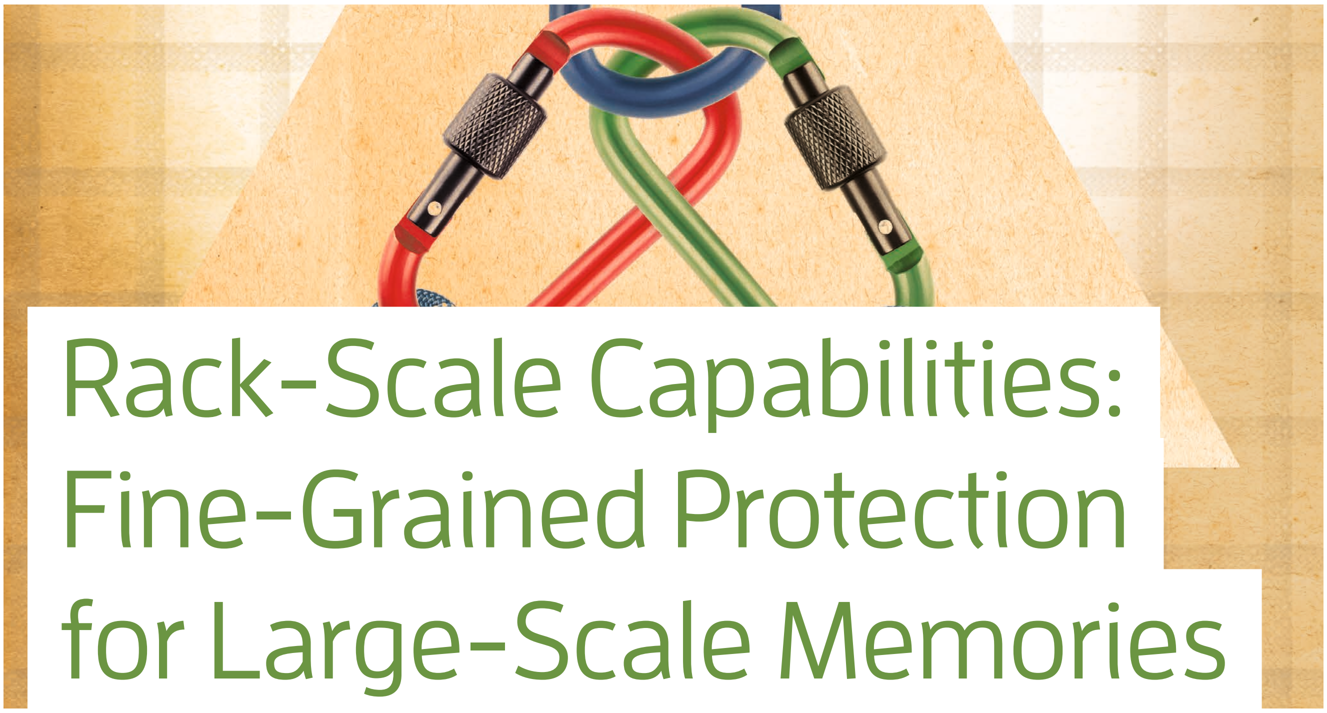


Rack-Scale Capabilities: Fine-Grained Protection for Large-Scale Memories

Kirk M. Bresniker and Paolo Faraboschi, Hewlett Packard Labs
Avi Mendelson, Technion
Dejan Milojicic, Hewlett Packard Labs
Timothy Roscoe, ETH Zurich
Robert N.M. Watson, University of Cambridge

Rack-scale systems with large, shared, disaggregated, and persistent memory need solid protection and authorization techniques. Our solution uses a memory-side capability enforcement processor that gates memory accesses through extended capabilities, enables fine-grained access control beyond a single address space, and minimally disrupts the programming model.

At the crossover point between technology adoption curves (such as persistent memory, rack-scale disaggregate memory, and memory semantics fabrics), it is vital to understand the benefits of defying conventions. In this article, we examine the ramifications of persistent fabric-attached memories (FAMs) at the rack scale and how approaches that predate memory paging, such as capabilities, could enable a robust and scalable protection mechanism.

Memory has always been considered a scarce resource that needs to be shared among multiple programs. Since the inception of virtual memory in the 1960s,[1] operating systems (OSs) have overcome the limitations of small physical memory by a variety of mechanisms to give users and programs the perception of unlimited memory. Virtual memory was—and remains today—a powerful mechanism with which to optimize memory allocation, simplify addressing, manage fragmentation, and

Digital Object Identifier 10.1109/MC.2018.2888769
Date of publication: 22 March 2019

allow oversubscription by means of paging out to slower media and paging in when needed. Over time, these operations have become so important that hardware support has appeared in all modern processors in the form of caches of the virtual memory tables, or translation look-aside buffers (TLBs).

Once the OS and hardware manage virtual memory as fixed size pages and include all of the necessary structures for virtual-to-physical address translation, it becomes natural to extend these tables to capture related concepts, such as access protection. For security, privacy, and error-containment reasons, not all programs are allowed to access all memory pages. Access-right information (typically read, write, or execute privileges) is stored as metadata associated with a page, cached in the TLB, and checked at page granularity. This arrangement makes the protection check fast because it is part of the translation process of each memory-access instruction. It also enables precise exceptions to generate accurate notifications about the nature of the protection violation. This is important to implement functionality, such as on-demand paging, shared libraries, or copy-on-write, and it requires the ability to precisely restart a faulting memory instruction after an exception occurs. However, it is a compromise because programs would naturally like to expose a different, often finer, granularity protection, possibly at the individual-object level, and not be tied to an arbitrary page size.

Page-level protection also creates the opportunity for malicious exploits, such as buffer and stack overflows, when multiple tenants share a single page or execute code in a shared library. Because all addresses within a page inherit the same protection, privileged execution of code may be achieved simply by gaining access to a piece of the memory address space. For small page sizes this has (historically) been considered an acceptable compromise across security, performance, and hardware complexity considerations.

However, the technology has substantially changed. Individual computers can afford terabytes of physical memory, and rack-scale systems federating hundreds of elements are approaching petabytes. At this scale, organizing memory in kilobyte-sized pages requires billions of pages, and the overhead to manage the mappings does not scale, as page tables and page-table walks overflow TLBs and caches. At the same time, this abundance of memory removes the original motivation for virtual memory (paging to disk), and most programs keep all of the data in memory because of performance issues. As a consequence, the trend is to shift toward large (1–2 MiB) or huge (1–4 GiB) page sizes, so that applications can allocate most of their working set right away, allowing page-table overhead to scale with memory-size growth and move the OS overhead out of the way.

Unfortunately, larger pages increase security risks and can expose the page to errors (or malicious attacks) because any address within a page can be accessed without additional fine-grained control. More importantly, the underlying problem comes from the bundling of the two key memory concepts, translation and protection, in the same page structure. This is becoming a primary cause of tension in the OS: the needs of a large translation unit and a small protection granularity are fundamentally incompatible, and we need a different approach. In addition, certain workloads experience huge performance benefits from superpages, but others suffer—in practice, the shift is toward greater flexibility in the size of translation units within the constraints of different page sizes.[2]

Paging was not the only memory protection concept developed in the 1960s. Segmentation and capabilities[3] are two alternative approaches that support variable-size memory units, from a single byte to the whole address space. Both approaches can coexist with paging, and, for a while, some processors supported segmentation, while other systems supported capabilities.

Capabilities are particularly relevant to this discussion: they are unforgeable tokens of authority used to protect memory at a fine granularity, down to a single-byte location. For a full implementation, they require processor instruction set architecture (ISA) support to keep extra information (hidden to application programming) associated with memory addresses (i.e., pointers) stored in registers and memory. A capability-enhanced CPU can check this information upon every individual memory access to ensure that the access is allowed. The check can be extended to manipulate the capabilities themselves, such as securely storing (and retrieving) them to memory, while preventing access from unauthorized code. ISA-supported capabilities can be passed in user space without performance costs, but they require invasive hardware changes to the memory hierarchy, the microarchitecture (e.g., extending the register file and caches to store the metadata), and the ISA itself. The software stack also needs to change to maintain and utilize capabilities effectively when describing data and code structures.

CHERI CAPABILITIES

Capability Hardware Enhanced RISC Instructions (CHERI) (www.cheri-cpu.org) is an example of an ISA-supported capability implemented as extended, or safe, pointers[4] compatible with off-the-shelf software. Simple pointers are references to memory locations, and they contain (virtual) addresses. Capabilities are extended pointers that contain base, offset, length, and protection bits (Figure 1). *Length* defines the address range that the capability can access, counting from the *base* address. *Offset* represents the individual memory access target (the virtual address to access memory through the capability is base plus offset). The protection bits grant read, write, and execute permissions. A process owning a capability can derive other capabilities with reduced rights, in terms of space or access. This allows a process to subdivide a capability to provide access to only a subset of the initial address range, or to remove rights, such as execute or write (maintaining the monotonicity of capability derivation). The software tool chain (compiler and linker) and programmers can selectively manage access to memory regions by passing to other programs capabilities that refer to a region subset or limited access rights. For example, a memory manager can hand out access to parts of the memory buffer to clients, or a server can provide write access to only a single writer, while allowing other clients only read access.

Supporting capabilities requires changing the ISA and microarchitecture. CHERI extends capability registers to access memory and adds the supporting enforcement logic. Enforcement compares the contents of capability registers with the attempted access after the capability has been manipulated through typical pointer arithmetic operations. Capabilities are enforced on data access to support passive data checks and instruction execution (e.g., procedure call/return, jumps) for active objects and compartmentalization. CHERI also adds privileged instructions to store/load to/from memory using capabilities. To prevent processes from forging capabilities stored in memory, a tag bit is maintained for each capability in memory, which is propagated through caches and the TLB into the capability registers. Any attempt to modify a memory location containing capabilities by unauthorized code clears the capability bit and effectively invalidates the capability, preventing it from accessing data. The tag enforces noncorruption and ensures valid provenance.

CHERI capabilities double the size of pointers from 64 to 128 bits (plus one tag bit). Using capabilities on a single node requires small changes to software and limited changes to the OS. Most changes can be hidden in libraries or directly implemented by the compiler and tool chain.

CAPABILITY ENFORCEMENT ACCELERATORS

CPUs and ISAs are evolving slowly. It takes several years for a new ISA feature to be implemented and even longer to reach the market, be supported by an industry-standard OS, and, finally, be adopted by application developers. ISA-supported capabilities are no exception. Moving some of the ISA support into a separate system (outside the CPU) could lower the adoption barrier.

Furthermore, ISA-supported capabilities exist within a single virtual address space. Sharing across address spaces (or persistent memory) requires additional OS support and incurs performance costs in crossing OS boundaries. This eliminates the performance advantage of ISA-supported capabilities in the user space when dealing with multiple processes or OSs at the rack scale.

The alternative to CPU-supported capabilities is a dedicated external component. In this case, we propose a memory-side capability-enforcement processor (CEP), a hardware controller (also called an *accelerator*) interposed on the load/store path between the CPU and the memory. The CEP acts as

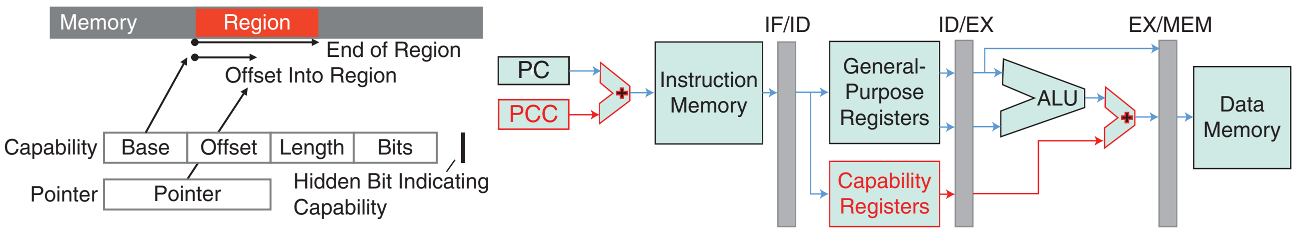


FIGURE 1. The format of CHERI capabilities compared with a simple pointer and (micro)architecture changes. IF: instruction fetch; ID: instruction decode; EX: execute; MEM: memory access; ALU: arithmetic logic unit.

a secure memory controller, taking the responsibility of guarding the access to the memory it controls through a capability system. The CPU can use it by issuing specific CEP instructions to access memory or to manipulate the CEP-stored capabilities. In its straightforward implementation, the CEP can be used to replace the ISA support for capabilities, with minimal changes to the rest of the system. However, some functionality, such as compartmentalization, may require the CEP to rely on OS support. The CEP also provides support for capabilities not covered by the ISA capability model, such as memory sharing (intra- and internode) and persistent memory.

Figure 2(a) shows how ISA-supported capabilities enable secure access within individual virtual address space. Applications can share memory, but capabilities cannot be stored in shared memory, nor can they be securely used to access shared memory. An ISA-based system cannot enforce capabilities across different virtual address spaces or different OS instances. The CEP overcomes these limitations, because it operates memory-side on the physical addresses, rather than virtual addresses, and introduces handles in the user space [Figure 2(b)]. The CEP tracks the handles and checks them when data are accessed so that only allowed processes can proceed. The CEP [Figure 2(c)] can also supplement ISA capability enforcement across virtual address spaces.

RACK-SCALE SYSTEMS AND CAPABILITIES

Enhancements in optical interconnects, memory semantics protocols, and the emergence of fabric-attached nonvolatile memory (NVM) are making rack-scale memory a reality. This enables the individual nodes in a rack-scale system to access all memory through a familiar load/store interface, with performance comparable to that of local memory access. The abundance of globally addressable memory enables new in-memory algorithms and non-partitioned data structures that are impractical on traditional clusters due to performance, power, and cost limitations. Unfortunately, it also further widens the chasm between protection and translation, making the case for capabilities even stronger.

The concept of capabilities needs to evolve to support memory in rack-scale systems with many nodes running independent OSs. When rack-scale systems also include shared NVM, as some emerging paradigms combining memory and storage advocate, capabilities need to evolve accordingly.

Rack-scale systems consist of multiple nodes, each running its own OS instance in support of the scale-out model. They also have a stronger trust

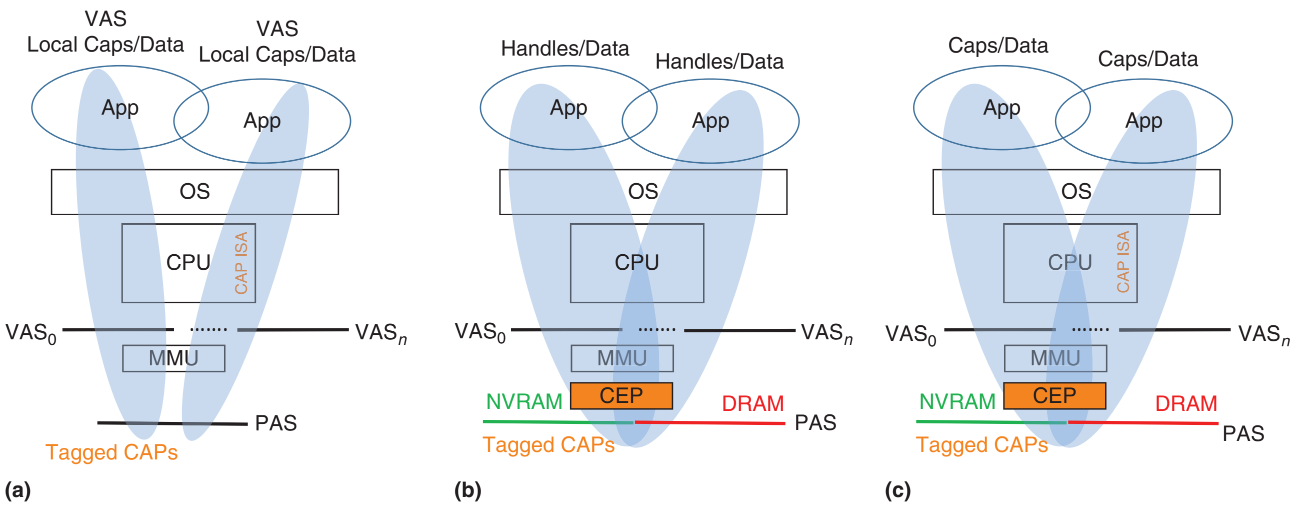


FIGURE 2. The CEP. (a) ISA capabilities allow fine-grained protection within a single virtual address space. (b) Transition: the CEP fine-grained protection uses handles across the physical address space. (c) Vision: the CEP supplements the ISA in fine-grained protection across VAS/PAS and NVM. DRAM: dynamic random-access memory; NVRAM: nonvolatile random-access memory; VAS: virtual address space; PAS: physical address space; MMU: memory management unit; CEP: capability enforcement processor.

model—if a single OS is compromised, the node boundaries prevent propagation to other nodes. In a distributed multi-OS environment, revocation of capabilities becomes a complex task because we can no longer rely on a single OS (and single execution hardware) to have full control of a capability. Managing distributed capabilities requires the careful interaction of hardware support, OSs, and the application runtime.

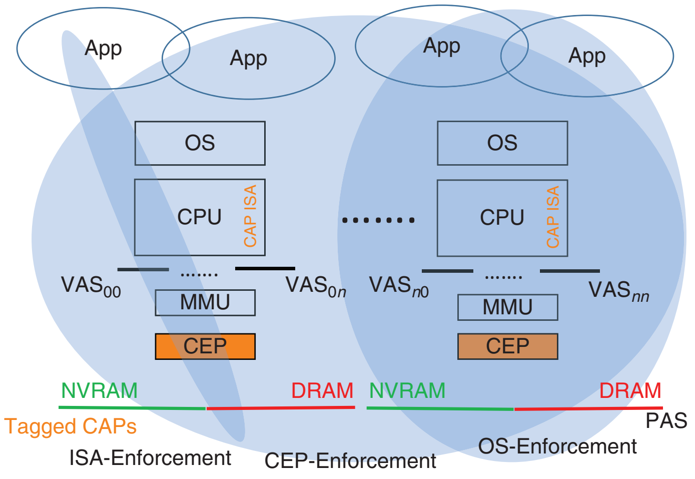
An interesting programming paradigm of rack-scale systems organizes applications into microservices and containers. They benefit from fine-grained protection because they can be packaged much more densely than what a given page size allows. In addition, they benefit from both code and data protection by selectively allowing which components can be invoked from other components. Delegation in the case of microservices is a very powerful programming approach to selectively enable access to individual components of the data structure at fine granularity.

In this environment, threats can come from a compromised or buggy OS, application, or any other piece of system software. The major concern is with unauthorized writing to the memory. When memory is persistent and not cleared after reboot, the threats/bugs are exacerbated because contents may persist beyond the lifetime of the OS. To address these threats, we leverage different security models. ISA support deals with the individual virtual address space; OS capabilities enforce a node-level trust model; and at the rack-wide scale, we leverage the TOR manager, secure enclaves, and the networking components that enable access to the FAM.

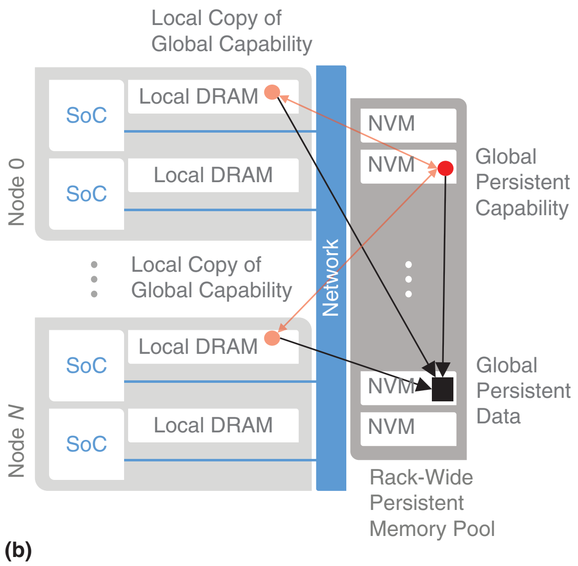
The CEP model naturally expands to cross-node capabilities in rack-scale systems. Because the CEP resides close to memory, it is effective in enforcing policies and management of the data access from multiple nodes, following the self-protecting memory principle. Although the performance implications of checking accesses for very fast (node-local) memory would be severe, they become tolerable for slower devices (NVM byte-addressable technologies) or when the accesses traverse a multihop fabric (FAM at the rack scale).

Another way to look at this is from the perspective of address spaces. ISA-supported capabilities take a virtual-address-space view [Figure 3(a), left], and an OS takes the node view [Figure 3(a), right]; the rack-wide view addresses the rack scale because any part of the NVM could be mapped into a single node. Because of the size of FAM and the distance from each CPU, we can offload some of the capability enforcement from the CPU into accelerators closer to FAM [Figure 3(a), center].

Figure 3(b) presents a sample rack-scale configuration that uses FAM



(a)


Local Copy of Global Capability
Node 0
SoC
Local DRAM
SoC
Local DRAM
Local Copy of Global Capability
Node N
SoC
Local DRAM
SoC
Local DRAM
Network
NVM
NVM
NVM
NVM
Global Persistent Capability
Global Persistent Data
Rack-Wide Persistent Memory Pool

(b)

FIGURE 3. Rack-scale capabilities. (a) Approaches to capability enforcement in rack-scale systems with fabric-attached memory: ISA-, OS-, and rack-supported. (b) Capabilities in a fabric-attached memory. SoC: system on chip.

pooled and accessible by all nodes, as some approaches advocate. Similar considerations apply if rack-scale memory is distributed and accessible by more traditional mechanisms, such as RDMA remote direct memory access or NVM express over fabrics. Adapting capabilities to the rack-scale environment is critical to reliable software development in that environment, but it also requires extending our notion of the model and implementation of the underlying capabilities. There is always a lowest layer of the system software (kernel, supervisor, hypervisor, whatever runs on the TOR control processor, and so on) that multiplexes the machine, and this is where the software that controls the lowest level of capabilities lives, the rack-wide capability management system. Anything else (virtual machine, containers, bare-metal OSs on a secure partition of the hardware) are above this layer.

Compared with single address-space capabilities (such as CHERI), which live and die with the creation and termination of a process, capabilities in a rack-scale system are long lived. They can outlive not only the process that created them or was using them but also an OS reboot or even reinstall. Capabilities can be stored in the memory of other nodes or in globally shared NVM that is saved across OS reboots. There is a temporal aspect of capability persistence that does not exist with ephemeral capabilities (local capabilities that live in local memory and a single process). In addition, when a capability is stored in persistent data, the capability itself has to be persistent for the system to be consistent. Because the notion of persistence is always tied to a certain class of failures, capabilities can be considered *persistent* when they are stored in nonvolatile device or anywhere outside the failure domain of the process that created them. Capabilities derived from a persistent capability can be ephemeral [e.g., they live in memory that disappears with the process, like local dynamic random-access memory (DRAM)], but the master capability needs to be persistent. The opposite is not true: persistent capabilities cannot point to process-local (volatile) memory; only ephemeral capabilities can.

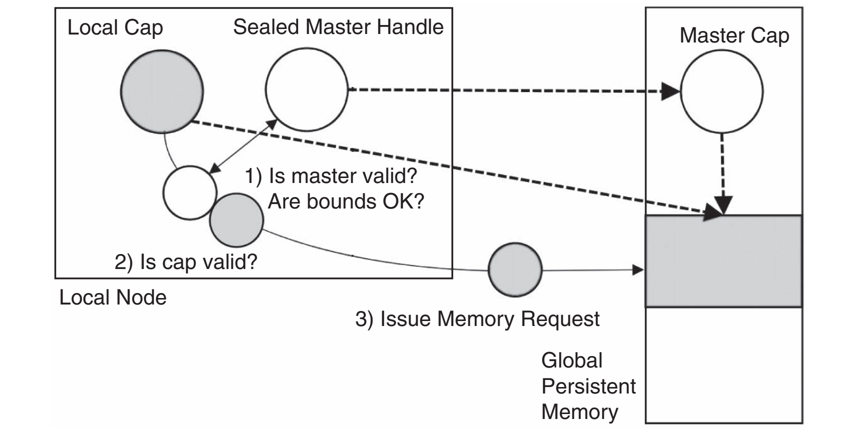
CAPABILITIES CAN BE STORED IN THE MEMORY OF OTHER NODES OR IN GLOBALLY SHARED NVM THAT IS SAVED ACROSS OS REBOOTS.

Local pointer bugs may corrupt local data within a process, but the corruption is limited to the process lifespan. With NVM, pointer bugs may persist in memory indefinitely, leading to corruption, regardless of program restart or system reboot, making fine-grained pointer and memory protection essential to the success of NVM-based systems for nonmanaged languages (and for the runtimes of managed languages, frequently implemented in C/C++).

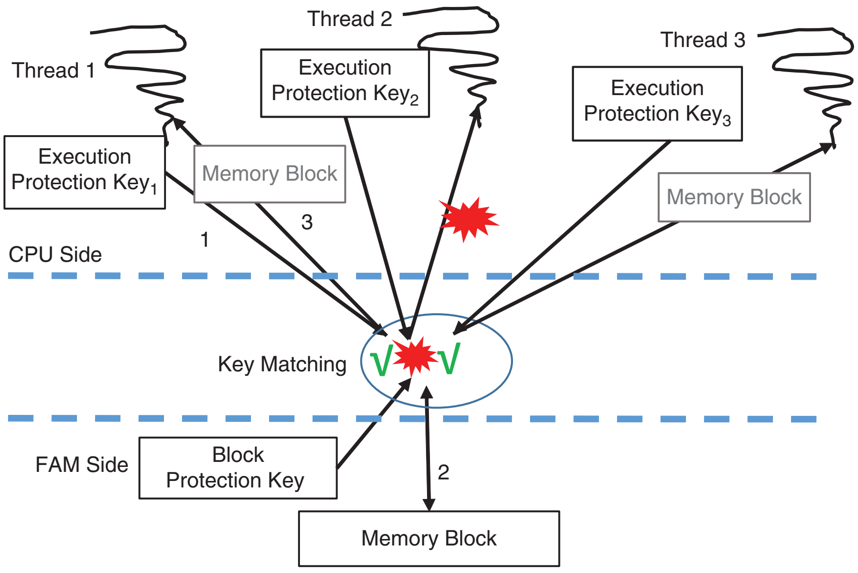
Unlike local capabilities, rack-scale capabilities can be named and accessed globally from any node, not just from the node where they were created. To accomplish this, we record the creation source node in the capability, so that accessing memory can be appropriately directed. Similarly to persistence, a capability pointing to global data has to be global [Figure 3(b)]. When global capabilities are passed to other nodes, revocation complications arise.

Rack-scale systems typically involve additional levels of memory translation beyond the processor's memory management unit. In addition to virtual (unique to a process) and physical (unique to a node) addresses, memory locations have a unique fabric address. These can be made of node identifier and local addresses (for legacy networks) or built in the protocol itself (for new interconnects, such as Gen-Z). Regardless of the mechanism, fabric addresses are larger than individual node addresses, and, ideally, one would like to have a direct translation from 64-b virtual addresses to unique fabric addresses. However, when using ISA load/store instructions, the smaller physical address (lower than 52 bits today) gets in the way, causing a disconnect between the CPU and memory, resulting in the need for memory-side translation support (which makes a CEP approach even more appealing).

Pursuing this kind of work requires the intersection of many areas of computer science. We six coauthors come from diverse and complementary backgrounds: microarchitecture, architecture, distributed systems, system software, and security. This makes us ideal collaborators



(a)



(b)

FIGURE 4. Approaches to revocation. (a) Redirection. (b) Key-based revocation.

and ensures that all of the aspects of the design are discussed and covered.

CAPABILITY REVOCATION

Being unforgeable, capabilities can be passed around to access resources (i.e., memory). In a rack-scale system, the other processes can be microservices that execute on other nodes through a distributed application programming interface. This creates an interesting complication: once the owner releases a resource, all of the capabilities representing that resource need to be revoked. Otherwise, a subsequent access will result in an error that would be difficult to debug and would require complicated client-side error handling schemes. It would be as if someone decided to change the lock to a shared closet, without telling everyone with a key that the key no longer works.

One-sided revocation is nontrivial in rack-scale systems because capabilities can be dispersed, and it may take time, and a complicated distributed algorithm, to reach revocation closure. For nonarchitectural capabilities, the OS maintains data structures that track trees of derived capabilities, which are then parsed to revoke all derived capabilities. Even in a single system, revocation represents a complex activity that can cause performance penalties and is nontrivial to implement efficiently. In CHERI, the locations of capabilities can be tracked with assistance from the paging mechanism, but this requires sweeping through the memory with suitable atomicity properties. In a rack-scale system, with distributed state, revocation is extremely complex and must avoid the need for global operations to ensure adequate scalability and reliability.

An alternative is lazy revocation, which can be accomplished by extending derived capabilities with copies of a master capability representing the same memory. On revocation, the master capability and the memory it represents are both freed. On each access to memory using other copies of revoked capability, a verification is first performed to determine whether the master capability is valid, followed by verification of the access right to memory. These two verifications can be conducted in parallel and be hardware accelerated [Figure 4(a)]. Another approach is to associate blocks of memory and threads accessing memory with matching keys. Upon each memory access, keys are matched using hardware.[5] If there

TABLE 1. Different approaches to capabilities.

	Features					
Approach	**Example systems**	**Distribution**	**Persistency**	**Revocation, GC**	**Granularity**	**HW/SW support**
HW ISA support	CAP, Plessey System 250, StarOS, IBM/38, iAPX432, Hardbound, low-fat pointers HW, CODOMs, M-Machine, and CHERI	Single process except Plessey System 250, StarOS, and iAPX, which are multinode	No support in StarOS, Hardbound, low-fat pointers HW, CODOMS, and CHERI	Revocation in IBM/38, CODOMS, M-Machine GC in StarOS, and M-Machine	Fine	HW/SW, ISA, OS, microcode, and compiler
OS	Mach, Chorus, Amoeba, KeyKOS, EROS, L4, Barrelfish, and Composite	Multinode clusters except for KeyKOS (multiprocess) and EROS, and L4 (1 node)	Capability to pager (Mach, Chorus, L4), FS (Amoeba), and VAS (KeyKOS)	Revocation: yes, except Amoeba and KeyKOS GC: no, except L4 and Composite (ref cnt)	Page, objects, and exceptionally fine	OS support and MMU
Languages and fat pointers	E, Joe-E, Caja, SoftBound, CCured, low-fat pointers SW, and Cyclone	Single process	No	No revocation GC optional	Objects	Language runtime and compiler

HW: hardware; ref: reference; SW: software; cnt: count; FS: file system; GC: garbage collection.

is a match, access is allowed, and if not, an exception is raised [Figure 4(b)] and communicated back to the application. The application can rerequest the capabilities and reissue the access (if it still has permissions to the memory region), or it can signal a protection violation to the end user.

Implicit in lazy revocation are two important points: software needs to react to traps caused by overrevocation and reacquire underlying capabilities with new keys, and a genuine protection fault is likely to be caused by a bug or a malicious exploit attempt. A well-behaved application should not try to access memory after a revocation, so the protection mechanism is a backstop and hopefully is rarely invoked.

IMPLEMENTATION ASPECTS

Historically, the primary challenge in scaling capability-based systems was revocation. Deriving capabilities results in chains that need to be torn during revocation. This is costly in single-node systems and unacceptable at the rack scale. The lazy approach we introduced addresses this challenge. Capabilities are invalidated, and verification is conducted every time capabilities are used. Memory-side accelerators allow verification at memory access speed. The performance of the CEP is affected by the number of capabilities, which can be cached by the CEP if needed. Capability-based fine-grained memory protection fits well with policies for elastic scaling of memory regions, enabled by splitting and merging of capabilities and corresponding memory regions.

To extend trust among the nodes, we need to rely on a secure and scalable memory fabric that supports managing capabilities. New interconnect standards, such as Gen-Z (https://genzconsortium.org/), extend memory semantics across nodes within a rack and also provide basic support for copying capabilities around through privileged operations.

OTHER APPROACHES TO CAPABILITIES

There is a rich history of capabilities, which can be classified as hardware, OS, and language supported (see Table 1). Only hardware-supported fine granularity and persistency, for example, CAP, StarOS and IBM System/38 (see Levy[3] for details). OS-supported, but not rack-scale, systems targeted clusters (e.g., L4,[6] KeyKOS,[7] Barrelfish[8]). Language-supported approaches are more flexible but have lower performance. They rely on objects within a single process, for example, low-fat pointers,[9] SoftBound,[10] and CCured.[11] Recently, vendors, such as Intel, introduced limited support for fine-grained

APPLICATIONS AND USE CASES

The Machine Research Program at Hewlett Packard Labs proposes a so-called memory-driven computing approach spanning from embedded through exascale computing. Hewlett Packard Enterprise recently demonstrated a rack-scale prototype of 160 TiB of memory attached to an optically connected memory semantics fabric. This prototype crosses an interesting threshold, offering significantly more memory than is addressable either by the physical addressing of industry standard architectures or the virtual addressing of OS kernels. Although this was designed as a testbed for hardware, firmware, and OS investigations, the prototype has also afforded the opportunity to explore applications of rack-scale systems and how capabilities can enhance those applications. Two are briefly described here.

PETA-SCALE TIME-VARYING GRAPH DATA STORES WITH MULTIPLE ACCESS ROLES

A huge variety of problems arising from the study of complex economic, ecologic, and biologic systems are most naturally represented as graphs. The efficient algorithms of graph theory can find hidden correlations and allow us to make inferences from incomplete data as long as we can efficiently manipulate both the graph and its associated metadata. This is where conventional scale-out systems are challenged, since the data distribution, caching, and prefetching algorithms can be rendered ineffective by the random nature of the underlying relationships. Even if care is taken to optimally partition a graph for a given access pattern, as the graph varies with time, the partitioning rapidly becomes inefficient. If the access pattern is random, the vast majority of the accesses are remote, thus preventing any effective use of locality. The memory-driven organization of the machine rack-scale infrastructure allows us to hold graph and metadata in a single shared memory pool, allowing distributed applications to access them at a fine, byte-level granularity. The use of graph theory across a longitudinal data set naturally invites multiple access roles: analysis versus evolution of the graph either with or without the metadata. Capabilities enable enforcement of roles, which can survive and be revoked independent of the execution lifecycle of any particular process or the underlying OS.

HARDWARE/APPLICATION COMPOSITION WITH ACCESS TO DISAGGREGATED PERSISTENT OBJECTS

There is an interesting intersection between capabilities and the emerging category of composable hardware, which today involves composition of storage and networking with fixed, relatively stateless CPU and memory resources. Container-based application development and rack-scale infrastructure allow for the low-level commissioning of just the right hardware, inclusive of accelerators and memory, for a particular container. Add in persistent objects in disaggregated memory, inclusive of data, applications, libraries, and you can gain the ability to remove a majority of spin-up/spin-down time and replace virtualized input–output operations with much higher-performance-shared memory operations. Capabilities allow all of those fabric-attached memory accesses, both sequential and simultaneous, to be authenticated and protected against errors while still allowing immediate access to in-memory objects as soon as fabric connectivity is established.

memory protection, such as MPX. For additional discussion on use cases, see "Applications and Use Cases."

We motivated the need for rack-scale capabilities as a consequence of increasing memory capacity paired with fine-grained (load/store) access to FAM. We described how rack-scale capabilities are evolving from traditional ISA- and OS-supported capabilities. We discussed the CEP as an alternative (or supplement) to ISA support. Finally, we described capability revocation as a key challenge and presented two solutions for hardware support for revocation.

Many challenges remain for a future work on rack-scale capabilities. ISA support is not extensible to the rack scale. Memory mapped from the FAM on one node may end up at different virtual addresses on other nodes. Self-referenceable structures or sophisticated ways of translating from virtual to physical to rack-scale address spaces become necessary. ISA support for capabilities is a long-term evolution, requiring more than five years to adoption. Providing similar functionality closer to FAM offers a faster pace of evolution and a more scalable and reliable solution. In addition to hardware, changes to the system software are required to support legacy applications. New classes of applications will evolve to fully utilize the benefits of memory-driven computing: load/store semantics and latency in accessing rack-scale fabric-attached NVM.[12]

We see many opportunities for deeper integration of hardware architecture, OSs, and programming models. The key technical question is how to balance the support across these three levels to achieve the desired performance, security, and flexibility. C

REFERENCES

1. F. J. Corbato and V. A. Vyssotsky, "Introduction and overview of the Multics system," in *Proc. Fall Joint Computer Conference (AFIPS '65)*. New York, 1965, pp. 185–196.
2. D. Milojicic and T. Roscoe, "Outlook on operating systems," *IEEE Comput.*, vol. 49, no. 1, pp. 43–51, Jan. 2016. doi: 10.1109/MC.2016.19.
3. H. M. Levy, *Capability-Based Computer Systems*. Newton, MA:

ABOUT THE AUTHORS

KIRK M. BRESNIKER is a fellow and chief architect of systems research at Hewlett Packard Labs. His research interests include novel hardware and software system designs. Bresniker received a B.S. in electrical engineering from Santa Clara University. He is a Senior Member of the IEEE. Contact him at kirk.bresniker@hpe.com.

PAOLO FARABOSCHI is a fellow at Hewlett Packard Labs. His research interests include intersection of architecture and software. Faraboschi received a Ph.D. from the University of Genoa, Italy. He is a Fellow of the IEEE. Contact him at paolo.faraboschi@hpe.com.

AVI MENDELSON is a professor of computer science and electrical engineering at Technion. He earned his Ph.D. from the University of Massachusetts at Amherst. His research interests include computer architecture, operating systems, reliability, cloud computing, and high-performance computing. He is a Fellow of the IEEE. Contact him at avi.mendelson@tce.technion.ac.il.

DEJAN MILOJICIC is a distinguished technologist at Hewlett Packard Labs. His research interests include operating systems, distributed systems, and systems management. Milojicic received a Ph.D. from the University of Kaiserslautern. He is a Fellow of the IEEE and was the 2014 IEEE Computer Society president. Contact him at dejan.milojicic@hpe.com.

TIMOTHY ROSCOE is a professor of computer science at ETH Zurich. His research interests include networks, operating systems, and distributed systems. Roscoe received a Ph.D. from the University of Cambridge. Contact him at timothy.roscoe@inf.ethz.ch.

ROBERT N.M. WATSON is a university senior lecturer at the University of Cambridge Computer Laboratory. He received a Ph.D. from the University of Cambridge. Contact him at robert.watson@cl.cam.ac.uk.

Butterworth-Heinemann, 1984.
4. R. N. M. Watson et al. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *Proc. 36th IEEE Symp. Security and Privacy*, May 2015, pp. 20–37.
5. R. Acherman, C. Dalton, P. Faraboschi, M. Hoffmann, D. Milojicic, and G. Ndu, "Separating translation from protection in address spaces with dynamic remapping," in *Proc. 16th Workshop Hot Topics in Operating Systems (HotOS '17)*, 2017, 118–124.
6. J. Liedtke, "On microkernel construction," in *Proc. 15th ACM Symp. Operating System Principles*, Copper Mountain Resort, CO, Dec. 1995, pp. 237–250.
7. N. Hardy, "KeyKOS Architecture," *SIGOPS Operating Syst. Rev.*, vol. 19, no. 4, pp. 8–25, 1985. doi: 10.1145/858336.858337.
8. A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, and S. Peter, "The multikernel: A new OS architecture for scalable multicore systems," in *Proc. ACM 22nd Symp. Operating Systems Principles*, Big Sky, MT, 2009, pp. 29–44.
9. A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proc. 2013 ACM SIGSAC Conf. Computer and Communications Security*, Berlin, Germany, 2013, pp. 721–732.
10. S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proc. 30th ACM SIGPLAN Conf. Programming Language Design and Implementation*, New York, NY, 2009, pp. 245–258.
11. G. C. Necula, S. McPeak, and W. Weimer, "CCured: Typesafe retrofitting of legacy code," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 128–139, 2002. doi: 10.1145/565816.503286.
12. P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic, "Beyond processor-centric operating systems," in *Proc. 15th Workshop on Hot Topics in Operating Systems (HotOS'15)*, Kartause Ittingen, Switzerland, 2015.

Digital Object Identifier 10.1109/MC.2019.2901913