ORC: Increasing Cloud Memory Density via Object Reuse with Capabilities

Vasily A. Sartakov
Imperial College London

Lluís Vilanova
Imperial College London

Munir Geden
Imperial College London

David Eyers
University of Otago

Takahiro Shinagawa
The University of Tokyo

Peter Pietzuch
Imperial College London

Abstract

Cloud environments host many tenants, and typically there is substantial overlap between the application binaries and libraries executed by tenants. Thus, memory de-duplication can increase memory density by allocating memory for shared binaries only once. Existing de-duplication approaches, however, either rely on a shared OS to de-deduplicate binary objects, which provides unacceptably weak isolation; or exploit hypervisor-based de-duplication at the level of memory pages, which is blind to the semantics of the objects to be shared.

We describe *Object Reuse with Capabilities (ORC)*, which supports the fine-grained sharing of binary objects between tenants, while isolating tenants strongly through a small trusted computing base (TCB). ORC uses hardware support for memory capabilities to isolate tenants, which permits shared objects to be accessible to multiple tenants safely. Since ORC shares binary objects within a single address space through capabilities, it uses a new relocation type to create per-tenant state using thread-local storage when loading shared objects. ORC supports the loading of objects by an untrusted guest, outside of its TCB, only verifying the safety of the loaded data. Our experiments show that, compared to hypervisor-based de-deduplication, ORC achieves a higher memory density with a lower performance overhead.

1 Introduction

In data centers, memory density determines how many applications can be deployed on machines with a given memory amounts. It is a critical factor impacting the cost of data centers, as memory leads to significant capital and operational expenses [3]. The problem of achieving high memory density is expected to worsen as applications move to larger working set sizes [20, 36], and machines encompass more memory to satisfy these applications [10].

Higher memory density can be achieved by *de-duplicating* memory pages that have the same contents across the constellation of virtual machines (VMs), containers, and processes running on machines. This exploits that, in practice, the same OS is used across VMs, the same applications across containers, and the same libraries across processes [7, 31, 41, 61].

We observe that there is a trade-off between the efficiency of de-duplication, and the achievable level of isolation between tenants. Containers and processes can achieve near-perfect memory density when they use a shared OS with mechanisms in binary loaders that explicitly identify de-duplication opportunities, e.g., through dynamic shared libraries [24, 25]. The high efficiency of de-duplication is due to the shared OS, which has visibility of memory at a *binary object level*. Cloud environments, however, require stronger isolation between tenants, i.e., by using VMs without a shared OS.

In contrast, hypervisors implement strong isolation at the instruction set architecture (ISA) level, moving OS-level semantics to the guest OSs. While this removes complexity from hypervisors, allowing them to provide strong isolation, it comes at the cost of losing semantic information about how memory pages are used by VMs for binary object allocation. Memory de-duplication must thus occur at a *page level*: the hypervisor compares page contents blindly across VMs and performs expensive page table manipulations when de-duplicating, both of which result in performance and tail latency overheads [8, 39]. While hypervisors can accept de-duplication hints from VMs to reduce the number of scanned pages [1, 31, 40, 51], this does not eliminate overheads.

Our goal is to design a new software stack for cloud environments that combines high memory density with low overhead by explicitly sharing memory at a binary object level, while providing strong isolation guarantees between tenants an relying on a small trusted computing base (TCB).

We describe **Object Reuse with Capabilities (ORC)**, a new layer in a cloud stack that extends a binary program format such as ELF [9] to eliminate de-duplication overheads across tenants with strong isolation and small TCB. ORC enables isolation domains to share binary objects, i.e., programs and libraries, explicitly. For strong isolation, ORC only shares immutable and integrity-protected objects. Object sharing is also always explicit, which eliminates the performance overheads that today's hypervisors introduce with blind page de-duplication. To keep ORC's TCB simple and small, object loading is performed by untrusted code, e.g., the guest OS. ORC extends the compiler and binary format to allow ORC-enabled objects to be shared at load time.

To design and implement ORC, we make the following technical contributions:

(1) Efficient isolation with sharing. Current cloud stacks are designed around the use of page tables to control isolation and sharing, but page table manipulation is expensive: inter-VM sharing requires exits into the hypervisor to modify nested page tables [60]; de-duplication must temporarily downgrade page table entries, which can severely affect tail latency [54].

Instead, ORC uses hardware support for *memory capabilities* [14, 16, 34, 64, 66] to isolate domains into *compartments*. Memory capabilities grant access to memory regions, can be copied between memory and registers, and are protected by hardware. They have been shown to be a building block for isolating cloud tenants with a small TCB by supporting an OS instance per compartment, as in today's VMs [49].

By using capabilities, ORC isolates multiple compartments within a single address space, while being able to share binary objects between compartments with virtually no overhead. ORC thus uses memory capabilities to isolate compartments within a single page table, and to enable them to safely and efficiently share binary objects in a controlled way.

(2) Sharing at object level. Current binary formats, memory layouts, and loaders are designed for sharing across address spaces. After an object is loaded into memory, formats such as ELF [9] assume that global variables are mapped at fixed addresses relative to the code. This is not a problem with per-process page tables, because an object's global variables are mapped to different physical addresses in each process. ORC uses a shared page table, which means that global per-process-and-object variables must be handled differently when sharing pages directly across compartments.

As a solution, ORC introduces a new type of variable relocation for *compartment-local storage* (CLS). ORC maintains absolute and code-relative references for code and read-only data, and the area for per-thread variables, i.e., thread-local storage (TLS). It also adds a new mechanism for per-process variables that replaces the traditional global variable references. This allows compartments to share immutable contents directly, i.e., code and read-only data, while still having per-process-and-thread state that is isolated across compartments. Under the hood, ORC allocates writable, global variables into each compartment's CLS, and loads objects to refer always to the executing compartment's CLS (similar to TLS).

(3) Untrusted loading of shared objects. When objects are shared across isolation domains, loading is typically controlled by the TCB. This object loading complexity bloats the TCB: it requires access to I/O devices, must load binary data into memory, and adjust memory contents to reflect load-time addresses, e.g., through relocations. Such functionality spans user-level, kernel-level and device driver code, and moving it into the TCB exposes a wide attack surface.

ORC avoids this issue by allowing untrusted compartments to handle most of the object loading process (i.e., storage and file system I/O, data processing and copying, and memory contents adjustments). When an object is requested for the first time, the untrusted compartment manages loading, and requests the TCB to register an immutable and integrity-protected version of the newly-loaded object. ORC's TCB verifies that the loaded object cannot be used to attack future compartments that reuse the same object, and makes it available to future compartment object load requests. The verification process is simple: it only requires (i) scanning the memory contents of the registered object to calculate a hash (used in future load requests to ensure object integrity), and (ii) ensuring that any capabilities it contains stay within the object and ensure its immutability.

Our prototype implementation of ORC includes a new compiler pass and loader support for compartment-local storage (CLS), a small TCB that manages compartments and enforces the properties for secure sharing of binary objects, and a port of a library OS and C standard library, executing on each compartment, that support various applications in ORC.

We evaluate ORC using three workloads: (1) we deploy a set of real-time video transcoding instances and compare different mechanisms for increasing memory density, showing that ORC performs equivalently or better than KSM while having a lower performance overhead; (2) we also use an in-memory key-value store to evaluate the impact of de-duplication mechanisms on tail latency, demonstrating the higher overhead of runtime mechanisms; and (3) we measure the performance cost of decomposition of applications into sharable compartments, show the scalability of ORC's isolation mechanism.

2 Increasing Memory Density in the Cloud

Memory density in cloud computing defines how efficiently the cloud provider is utilising memory. Improving memory density is crucial for providers, because memory is often the main resource that determines how many tenants can be accommodated [33]. While providers want to exploit as many memory-sharing opportunities as possible, they must ensure that tenants and their workloads remain isolated. We first discuss different approaches and their associated challenges for page-based isolation and memory sharing (§2.1). After that, we provide background on memory capabilities, which can act as an isolation mechanism without some of the drawbacks of page table-based isolation (§2.2).

2.1 Page-based memory sharing and isolation

Cloud tenants expect strong isolation for their applications and data from that of other tenants, while providers seek ways to minimise total physical memory footprint by finding shareable memory across tenants. The two goals are at odds with each other: the mechanisms we have for efficient memory sharing are, in essence, reducing the level of isolation between tenants. With today's isolation technologies, we must choose between containers [35, 38] and VMs [2, 30, 61], which in turn dictate the mechanisms used for memory sharing.

(1) OS-managed memory sharing. Container-based deployments rely on a shared OS for isolation. The OS provides user-space abstractions for sharing memory, and a loader can explicitly map the same binary object (e.g., dynamic library) across multiple processes. The OS has thus enough user-level information to de-duplicate object contents at load time, sharing memory across containers without additional runtime overhead.

Efficient memory density in containers comes at the expense of isolation. Containers are not considered as strongly isolated compared to VMs, because they are managed by a shared OS kernel. Such a large, shared TCB is too large and complex to eliminate all vulnerabilities [11], which can be exploited by a malicious container to access information from other containers and tenants [12, 13].

(2) Hypervisor-managed memory sharing. VMs offer stronger isolation compared to containers: they offer a narrow virtualization interface at the level of the ISA, with a potentially small TCB (the hypervisor) that makes security vulnerabilities less likely [29, 57]. To share memory between VMs, typical hypervisors such as ESX [61] and KVM [30] identify and eliminate redundant memory pages at runtime. The hypervisor lacks visibility into the semantics of user-space applications within each VM, so it must periodically scan the memory of each VM to find pages with identical content, and remap these guest physical pages across VMs into the same host physical page to de-duplicate their contents.

A popular implementation of this approach is Linux *kernel same-page merging* (KSM) [1], which the Linux KVM hypervisor leverages to eliminate duplicate memory pages across VMs. KSM periodically scans physical memory to find identical pages, and deduplicate them by mapping a single physical copy to multiple virtual locations. It also marks those pages as copy-on-write (COW), allowing the shared page to be duplicated before any modifications are made. ~~KVM computes a hash of page contents, and stores it in red-black trees to support content-based searches to find page duplicates.~~ KSM uses red-black tree structures to search for memory pages with identical content. ~~For efficiency, KSM keeps two such trees with page hashes: (1) a *stable tree* with already-shared pages, which are write-protected to support copy-on-write; and (2) an *unstable tree* with pages that are not shared but whose contents' hash has been computed.~~ For efficiency, KSM utilizes two separate trees: (1) a *stable tree* that contains already-shared pages, and (2) an *unstable tree* that represents pages not shared but scanned previously. During the scanning process of a memory page, KSM first searches for a match in the stable tree. If the page is found in the stable tree, the redundant copy is eliminated through merging. If there is no match in the stable tree, KSM checks whether the page has been modified since the last scanning round by comparing checksums. If the page has not been modified, it is considered a suitable candidate for searching in the unstable tree. If the page is found in the unstable tree, merging occurs, and the shared page is inserted into the stable tree. Otherwise, it is inserted into the unstable tree as a scanned page. The unstable tree is also reinitialized after each scanning round. ~~KSM periodically scans pages not in the stable tree, and computes their contents' hash to find new de-duplication candidates (pages must be temporarily write-protected to compute their hash). If the hash is already present in the stable tree, the page is de-duplicated. If the hash is already present in the unstable tree, for the same page, the page is promoted into the stable tree. If the hash is already present in the unstable tree, for another page, both pages are de-duplicated and promoted into the stable tree. Otherwise, the page is added to the unstable tree. KSM flushes the unstable tree every time it finishes scanning all pages.~~

Despite the advantages of hypervisor-based isolation, its memory de-duplication mechanisms introduces several challenges: (1) blindly scanning page contents and manipulating their permissions comes with an overhead on average performance and tail latency [8, 39, 54]; (2) hypervisors lack the visibility of a container's OS to application-level load-time semantics, and therefore must rely on page scans; (3) the use of larger pages in the cloud improves memory performance [27, 47], but can reduce memory density by making memory de-duplication less frequent; and (4) the use of copy-on-write on de-duplicated pages has been shown to be vulnerable to timing side-channel attacks across VMs [26, 45, 58, 59, 67, 69].

2.2 Isolation with memory capabilities

Using paging for both translation and protection introduces performance challenges in traditional virtualized environments, which are tied to the management granularity, as described above. In contrast, *memory capabilities* offer an alternative memory protection mechanism that is more flexible, robust, and efficient to manage. At the same time, memory capabilities can co-exist with the use of paging for translation [4, 6, 18, 19, 64].

Memory capabilities replace integer-type pointers with protected capabilities. Unlike integer pointers, capabilities provide information to enforce accesses within a given address range and access type. Capabilities can thus be used to partition a single address space into multiple, isolated memory

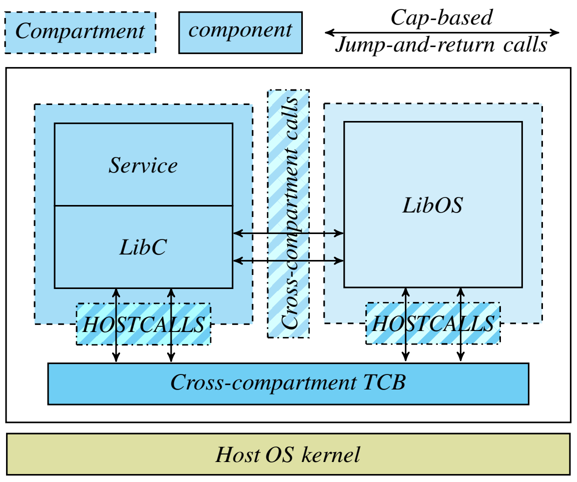


Fig. 1: Using capabilities to create compartments

regions, therefore allowing the use of a single page table across isolation domains.

Memory capabilities. The CHERI [62] architecture provides a modern implementation of memory capabilities. It introduces new instructions, registers, and other hardware primitives to support capabilities. CHERI enforces three properties: (1) *provenance validity* ensures that a capability cannot be created from an arbitrary sequence of bytes, and can only be derived from another capability; and (2) *capability integrity* ensures that capabilities in memory cannot be modified. *One-bit validity tags* distinguish pointers from integers; and (3) *monotonicity* ensures that a capability's permissions, including its bounds, cannot be expanded but only reduced.

CHERI can thus replace all pointers in an application with capabilities to precisely enforce the permissions of each memory access. This is known as the *pure-capability mode* (or *pure-cap* for short). Pure-cap enables spatial memory safety and fine-grained memory sharing, but requires ABI changes and other source code changes, e.g., to pointer-integer casts. CHERI also support a *hybrid mode*, in which code uses legacy, capability-unaware instructions. These are checked through a pair of implicit registers, the *program-counter* and *default data capabilities*, for code and data accesses, respectively.

Code can use the capability-aware `CInvoke` instruction to perform function calls across isolated domains, carrying the necessary capability arguments across domains.

Capability-based compartmentalization. Capabilities offer a good mechanism to isolate software components, and their flexibility and management efficiency can eliminate the overheads of page-based sharing. CAP-VM [49] proposes a new capability-powered compartmentalization mechanism for the cloud, which brings together some of the benefits of VMs and containers. Each capability compartment (cVM) has its own separate address sub-range within a shared address space, and executes programs in CHERI hybrid mode. cVMs are managed by a shared TCB component called the *Intravisor*, similar to a VM hypervisor. The Intravisor is a host process that starts cVMs as one or more host threads within its address space, using the default capabilities in CHERI's hybrid mode to mutually isolate cVMs. In turn, each cVM has its own library OS instance to support private namespaces and program execution environments.

Fig. 1 shows how capabilities can be used to create multiple capability compartments, potentially sharing binary objects across compartments. An Intravisor, or some other compartment TCB, can give each compartment the capabilities needed to jump into/call code in other compartments, effectively sharing binary objects if two or more components have capabilities to the same object. The figure also shows how capabilities can be used to request Intravisor operations too (*hostcalls* at the bottom).

Note that the above figure suggests that it is possible to share binary objects efficiently by compiling software components using CHERI's pure-cap mode. This is not possible today, because: (1) the Intravisor has no explicit information about the extent and shareability of objects; and (2) existing storage formats and memory layouts of pure-cap binary objects assume that each compartment has its own page table.

In particular, existing binary object standards, such as ELF [9], assume that global variables are reachable through constant addresses relative to code locations. If we use a per-process (or compartment) page table, we can physically share non-writable pages across processes, while having separate contents for writable pages. This is no longer the case if we use a single page table, which is the only way to avoid the page table management overheads identified above.

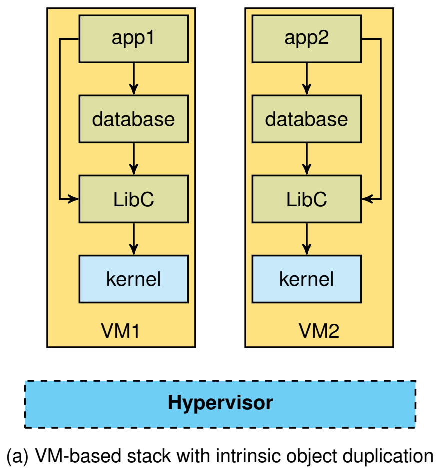
2.3 Efficiency and security considerations

The goal of this paper is to provide efficient memory density in cloud environments. To achieve this goal, our solution must fulfill the following requirements:

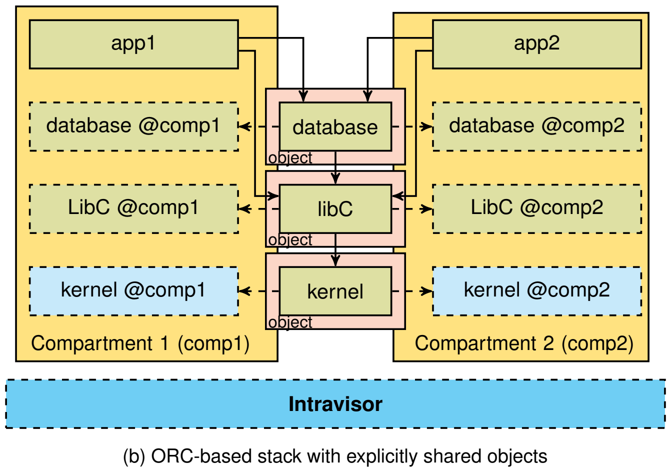
(1) Strong isolation with a minimal TCB: Memory sharing is needed for density, but it should not undermine isolation between tenants. We must thus reduce the attack surface by providing a small TCB with a narrow interface to manage isolation and sharing for density.

(2) Low performance overhead: The sharing mechanism should not incur high overheads in terms of CPU cycles, and it should not prevent the system from performing other optimisations, such as using large pages to reduce TLB misses.

(3) High sharing precision: The sharing mechanism should support arbitrary object sizes and have visibility into the intended object sharing semantics. An ideal solution should not miss opportunities to share, nor unintentionally share memory that soon diverges into different contents. This can be a problem with KSM, because it blindly de-duplicates pages solely based on content and access frequencies.



(a) VM-based stack with intrinsic object duplication



(b) ORC-based stack with explicitly shared objects

Fig. 2: Comparison between VMs and ORC compartments with explicit object sharing (Dotted lines show compartment-local variables.)

Note that sharing memory can be used as an unintended *side channel* across compartments. This is an intrinsic trade-off between memory density and isolation that all cloud providers face, regardless of the employed mechanism. Given the importance of side channels, there are proposals to avoid or mitigate them at both software and hardware levels [23, 44].

The sharing of binary objects in this work is limited to side channels on accesses to (i) code and (ii) read-only data only. Since the user controls which binary objects to share and when, they can decide on a suitable policy for trading off between memory efficiency and side-channel resistance. We leave the exploration of such policies to future work.

3 Design of the ORC system

In this paper, we exploit the capabilities implemented by the CHERI architecture [62] to implement both software compartmentalization and efficient binary object sharing.

ORC makes the sharing of binary objects explicit, so that the use of expensive physical memory can become denser without the performance overheads of de-duplication. Fig. 2 shows an example of this with two applications (app1 and app2) that use multiple object binaries that are identical across VMs, including the OS kernel (database, libC and kernel).

Fig. 2a shows our baseline system, in which each application is deployed in separate VM for maximum isolation. In this case, the hypervisor incurs typical overheads of memory de-duplication.

In contrast, Fig. 2b shows the approach taken by ORC. Programs are isolated into *compartments* – shown as light yellow boxes – which contain all the needed objects (app1, app2, database and libC) as well as their own OS kernel instance (kernel). ORC compartments are deployed using a shared page table, and obtain access to non-overlapping addresses using *memory capabilities*. Both the page table and capabilities are controlled by the TCB, shown as *Intravisor*.

Compartments are strongly isolated: each has its own OS instance, and are restricted to access non-overlapping memory address ranges. Sharing objects across compartments is supported through capabilities, which provide access to the object's contents (light red boxes with objects database, libC and kernel). If possible, the ORC program loader requests capabilities from the Intravisor for an object that has already been loaded by another compartment. Otherwise, the compartment loads the object itself and registers it with the Intravisor, so that future compartments can reuse it.

To make object sharing across compartments safe, the Intravisor must ensure that a shared object cannot be modified after registration. This, of course, implies that the registering compartment cannot change the object after registering it, but also that shared objects cannot contain writable state. We indicate this with the dotted lines in Fig. 2b: each shared object is recompiled to have per-compartment instances of any writable state. We refer to this as *compartment-local storage (CLS)*.

As a result, objects can be efficiently shared across domains, while retaining strong isolation down to the level of separate OS instances. Furthermore, sharing is part of the cloud software stack, ensuring that the memory density benefits do not come at the expense of performance overheads.

3.1 Architecture overview

Fig. 3 shows the high-level architecture of ORC. Programs execute within a compartment (shown as yellow boxes) and have statically and dynamically-linked objects, as usual.

❶ All potentially shareable objects, including the main program, must be compiled with our ORC-specific extensions. These extensions move all the writable state of an object into the CLS, i.e., all global writable variables, which is supported by extending the binary storage format and the loader (technical details below). The figure includes an example with three global variables, a constant, a thread-local, and

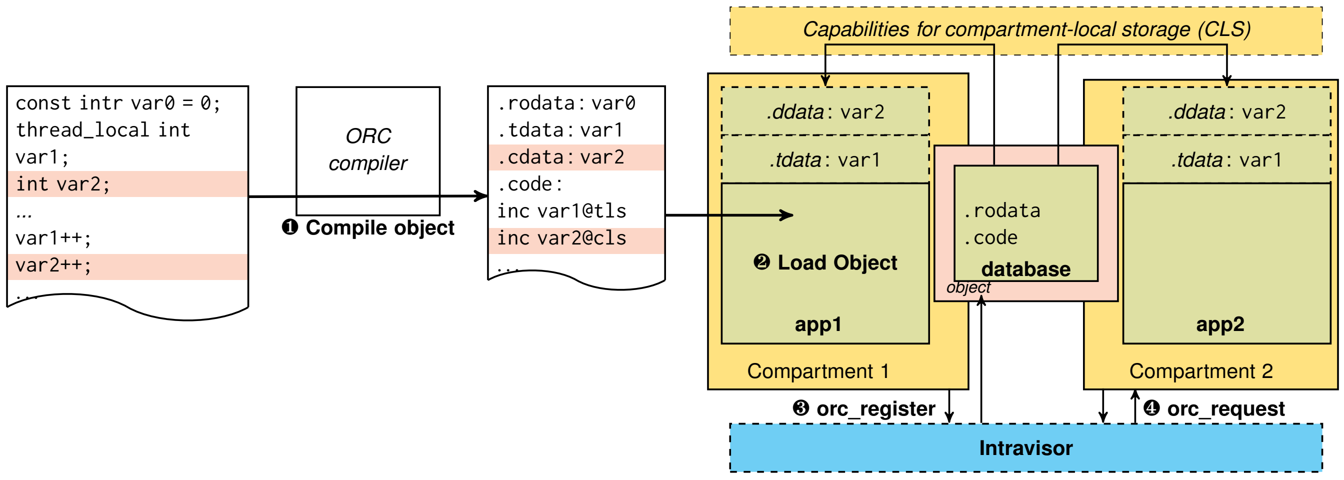


Fig. 3: Architecture of the ORC stack (Circled numbers identify operations referenced in the text.)

a writable variable (`var0`, `var1` and `var2`, respectively). Of the three variables, only `var2` is moved to the CLS, because thread-local variables are already stored in a per-thread data structure (the TLS [17]).

Note that none of these elements are part of the TCB – compartments can still uses private copies of objects without the ORC extensions, and only references to global writable variables are changed to point to the CLS. Heap allocations are supported as usual, which results in private allocations to each compartment.

Each compartment has its own, untrusted loader (`ld.so` in Linux systems). The compartment itself therefore loads the required objects by reading them from storage and parsing their contents according to the binary format (e.g., ELF). ❷ When the loader finds an ORC shareable object, it allocates the necessary memory to load the object into memory and prepares it for execution.

❸ After loading, the compartment calls the Intravisor's `orc_register` operation to safely register the loaded object for future use. The compartment passes the capabilities to where the object's contents are loaded, and a list of variable references in the object's code. The Intravisor then copies the object to a new location controlled by itself, computes a hash of its contents, resolves the variable references to the new load address, and registers the object's hash and allocation capabilities for future use. At this point, all compartments, including the registering one, proceed in with knowledge that the shareable object is available in the Intravisor.

❹ When a shareable object is registered in the Intravisor, the compartment requests it via `orc_request`, passing it the expected object content hash. If the hash matches that of a registered object by `orc_register` above, the Intravisor releases the capabilities for that object. The loader then proceeds by allocating the memory for the CLS variables, and subsequently loaded objects can reference this one.

Note that the main program itself can be an ORC shareable object. In that case, after loading it with `orc_request`, the whole application is ready for execution.

3.2 Compiler support and binary format

To support shared ORC objects, the compiler extends the binary format with support for CLS variables. When the ORC options are enabled, the compiler adds a flag to identify the object as ORC-enabled, and moves every writable global variable to the CLS.

To support CLS, the compiler replaces each reference to a global, writable variable with a new relocation type. Such relocations are resolved at load time to point to the per-container instance of that variable (see below), similar to how thread-local variables are moved to the TLS at load time.

We can see this process on the left-hand side of Fig. 3. Code, constant variables, such as `var0`, and thread-local variables, such as `var1`, are handled as usual by the compiler, placed in the `.code`, `.rodata` and `.tdata` sections of the ELF object, respectively, generated with their standard relocations. Global, writable variables, such as `var2`, are placed in a new `.cdata` section, and references identified with the new `@cls` relocation.

3.3 Secure object loading and reuse

The compartment loader brings the object's file contents into memory, and handles all relocations that are independent of the load address. It then calls the Intravisor's `orc_register` operation by passing the capabilities that delimit the memory regions in which the object was loaded and a description of the yet-unprocessed relocations.

To ensure that the object contents cannot be changed once shared, the Intravisor allocates new memory in capabilities C, copies the object contents into them, and checks that the object contains no capabilities pointing outside the C allocations (to avoid a malicious use of `orc_register`). At this point, the Intravisor computes a hash *H* of the object contents in memory, resolves the remaining relocations that depend on the information of the secure load location, and registers the object's hash and a non-writable version of the allocation capabilities, *H* and *C*, respectively. The CLS relocations are replaced with a value that points to a per-compartment memory address that holds all CLS variables of that object (see §4).

When a compartment calls `orc_request` in the Intravisor, it passes the hash *H*. If an object with hash *H* exists in the Intravisor (e.g., it was previously registered with `orc_register`), the Intravisor returns the capabilities for it, which are ready to use by the calling compartment.

The object hash *H* is computed by the Intravisor before any location-dependent relocations, and so it is also known by the requesting compartment. If an object with hash *H* exists in the Intravisor, we know its integrity and isolation are ensured. The Intravisor does not ensure object correctness beyond relocation resolution, which should be handled through other means, e.g., attestation checks as part of software supply chains, for which hash *H* can be helpful.

4 Implementation

We implement ORC on the Morello platform, a development board from Arm that has support for the CHERI capability extensions. In this section we describe how we: build ORC compartments by extending CAP-VMs [49] with our own library OS to maximize object sharing; add CLS support through a new LLVM compiler pass; and implement the necessary logic to securely load shared objects.

Both our Intravisor and the host OS are implemented as hybrid capability code based on the existing CAP-VMs and CheriBSD [22] projects. We extend the Intravisor to support pure-cap compartments, i.e., using the pure-cap CHERI ABI, and the program loading operations, which adds 530 and 240 lines of C and assembly code, respectively.

For our evaluation, we also port the SQLite database [56], FFmpeg [21] with the `libav` libraries, and Redis [48] to support the pure-cap CHERI ABI and ORC. In total, the porting requires approximately 350 lines of code. Note that besides adding the system functionality specific to ORC, efforts went into porting code to the pure-cap model.

Library OS and standard C library. To increase object sharing across compartments, we make the library OS and low-level C library support a pure-cap build, as no such software components exist with the necessary functionality. We implement our own pure-cap library OS kernel, which is based on Unikraft [32] and CubicleOS [50], from which we use 40 system calls and 9,061 lines of code. We extend it with support for the CHERI ABI and add a capability-aware memory allocator.

We also use a pure-cap version of the C library for our evaluation applications. It is based on musl-libc [43], whose pure-cap support is maintained by Arm. Our fork has 494 functions and 19,717 lines of code. We modify it to introduce a capability-aware memory allocator based on `dlmalloc`, and a few extra changes for compatibility with our library OS.

Compiler support and CLS. We prototype our ORC compiler as an LLVM pass. It replaces all references to global, writable, non-TLS variables with a call to function `__cls_get_addr`, which returns the compartment-local version of that variable. Internally, `__cls_get_addr()` is implemented using regular capability-aware instructions (`cgetaddr`, `cincoffset`, `csetlen`, etc.). The function retrieves the address of the variable from the input capability and makes it relative to the beginning of the data section. After that, it applies the relative offset to the capability that points to the shadow data section. Finally, it creates the replaced capability by limiting its size to match that of the original capability.

The `__cls_get_addr` function works similarly to how TLS is supported, i.e., through `__tls_get_addr` in the TLS definition for ELF [17]. It takes the shared object's identifier (assigned at load time) and the variable offset within the CLS (assigned at compile time), and returns a capability granting access to calling compartment's copy of that variable.

For simplicity, our compiler pass inlines `__cls_get_addr` into the generated code – a production implementation should use a separate CLS relocation type – and it uses a TLS variable to point to the compartment's CLS buffer for that object. This means that the loader (described below) only needs to implement TLS variables, accessed through the `tp` register in Arm, to support both TLS and CLS.

Secure object loader. To simplify application deployment, we implement a loader that takes deployment scenarios, i.e., list of binary paths to load into memory. The program deployment logic loads binaries into the target memory regions, resolves relocations, and generates all capabilities needed in the PLT and GOT of a pure-cap program [63].

The deployment scenario also identifies shareable objects and provides their hash, so that they can be reused if previously loaded through `orc_register` and `orc_request`.

Discussion. Our current implementation showcases the core ideas of ORC, but has a few shortcomings that affect memory density and performance:

CLS performance. The current CLS implementation uses TLS variables for simplicity. This results in new capabilities fetched from the TLS and adjusted to the corresponding variable on every call to `__cls_get_addr` (except for reuse optimizations in the compiler). In a future version, we would consider pre-calculating the per-variable capability at dynamic link time, so that no new capabilities are created at runtime by `__cls_get_addr`.

CLS compatibility. Since we use a compiler pass, we cannot support pre-initialized variable references on other data structures, e.g., a statically-initialized array entry pointing to a CLS variable address. In our library OS (Unikraft), we found only a single place where this was necessary. Adding compiler support for new CLS relocations would solve this problem by adjusting data structure addresses at dynamic link time.

In addition, our `__cls_get_addr` implementation assumes a per-compartment CLS. We plan support per-process CLS, allowing for multiple processes within the same compartment.

5 Evaluation

We ask the following evaluation questions: (1) how efficient is ORC compared to KSM? (2) what is the impact of ORC on tail latency compared to KSM and (3) what is the execution and compilation overhead of ORC, as a function of the degree of binary object sharing.

5.1 Experimental set-up

Workloads. We evaluate a real-time video transcoding micro-service that scales out to a large number of clients. The micro-service uses *FFmpeg* [21] to perform transcoding. A single FFmpeg instance consumes a fraction of the CPU and memory resources on the machine, allowing multiple instances to be run. By de-duplicating memory, we can support more FFmpeg instances and thus more concurrent clients. We compare the deployment of the micro-service using ORC to one that uses Linux KSM as a baseline for memory de-duplication.

We also consider other workloads to evaluate specific characteristics of ORC: (1) we use Redis [48] with the *memtier* benchmark [37] to understand the impact on request tail latency of ORC and KSM; and (2) we use the SQLite database [56] and its *speedtest1* benchmark [55] to compare the performance of different object sharing scenarios.

Test-bed. We deploy ORC on a Morello board [42], which has an Armv8-A CPU with hardware support for CHERI [66]. The board has 4 CPU cores running at 2.5 GHz, with 16 GB of DDR4 memory (64 KB L1, 1 MB L2, and 1 MB L3 caches).

The experiments compare two OSs: (1) Ubuntu 22.04.1 LTS with Linux v5.15.0, which only runs native arm64 binaries with no CHERI support, and (2) Hybrid CheriBSD version 14 (release/22.05p1) [22]. The Linux OS is used to measure KSM, and can also run the entire ORC stack without isolation guarantees (i.e., disabling our compiler pass and eliminating capability management instructions in the Intravisor and loader). The CheriBSD OS is used to measure ORC with all its isolation guarantees, as described in this paper (all ORC results use CheriBSD unless otherwise noted).

Note that the same source code executes on all compartments, so that we can compare ORC and KSM despite the different compiler options and underlying OS support.

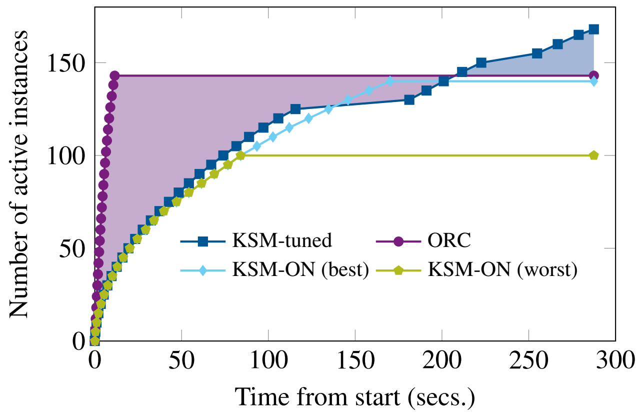


Fig. 4: Efficiency vs. overhead of de-duplication (FFmpeg). The shading indicates the difference between ORC and the best-performing alternative.

5.2 Efficiency and performance overhead

We now evaluate the trade-off between memory efficiency and performance overhead, comparing ORC and KSM. We deploy our transcoding micro-service, which can increase throughput (number of deployed instances) with higher memory density.

The experiment deploys new transcoding instances until it hits one of two limits: (i) memory limit – when the instances consume all available physical memory, but there are still spare CPU resources to support more instances; and (ii) CPU limit – when at least one of the instances can no longer transcode at real-time due to a lack of CPU resources.

Each micro-service instance has FFmpeg's main program and libraries, our library OS and the standard C library (see §4), and occupies around 111 MB of memory (11 MB in binaries with read-only variables, sharable by ORC, and 100 MB of heap). The transcoded video has a resolution and frames-per-second configuration such that, without de-duplication, the experiment hits the memory limit, while optimal de-duplication leads a higher number of deployed instances that eventually hit the CPU limit.

We deploy multiple KSM configurations with different de-duplication and overhead trade-offs:

KSM-tuned: This is the default policy of the *ksmtuned* daemon [70]. Every 60 secs, it checks the share of free memory, and starts KSM if it is below 20%. It also dynamically adjusts KSM parameters: the number of pages to scan on each iteration (`pages_to_scan`) is gradually increased when de-duplication speed is insufficient.

KSM-ON: This hand-tuned policy achieves the best memory efficiency for our workload, but consumes significant CPU resources. KSM operates constantly and assumes 20,000 `pages_to_scan`.

FFmpeg instances. Fig. 4 reports the number of active FFmpeg instances over time, under ORC and the various KSM

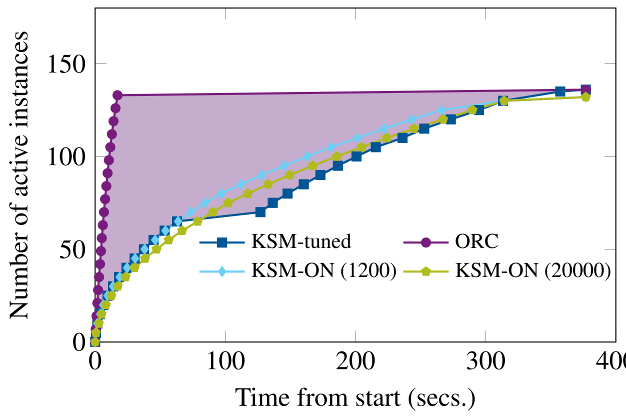


Fig. 5: Efficiency vs. overhead of de-duplication (large binary). The shading indicates the difference between ORC and the best-performing alternative.

configurations. Given the performance of a single FFmpeg instance, we have enough CPU resources for up to 180 instances, but only enough memory for up to 127 instances without de-duplication.

The results show that KSM-tuned first reaches the memory limit (127 instances) after 119 secs, and does not de-duplicate memory until 58 secs later. It then hits the memory limit again at around 227 secs, and after 23 secs is able to de-duplicate enough memory to deploy 180 instances.

In contrast, ORC is designed to get maximum density almost instantaneously, creating 142 instances in just 11 secs – this is a 20× speedup over KSM-tuned for the same number of instances. Note that the 142 instances deployed by ORC correspond to 11% more than the 127 instances we would get without de-duplication. This is because code occupies 11% of memory on each compartment (see above). KSM is able to de-duplicate further pages by also looking at all data contents.

The speedy deployment of ORC instances is critical for achieving high overall performance, measured by the number of processed images over time. Even if it deploys fewer instances over the long run, ORC outperforms various KSM-based policies by processing between 15% to 35% more images in a 300-second timeframe. To achieve the same number of processed images as ORC, KSM-tuned deployment requires an extra 141 seconds beyond its peak performance of 180 instances, while none of the other policies are able to outperform ORC since they never reach ORC peak performance.

The de-duplication rate of the default KSM policy is quite slow, as it minimizes CPU overheads by operating only under memory pressure (80%) and then at a limited rate of memory scanning (`pages_to_scan`). This can negatively impact environments where processes are frequently created and destroyed, so we also evaluate the case where KSM maximizes de-duplication rate with KSM-ON. KSM is probabilistic in nature, so we show the best and worst results of twenty different runs of the same KSM-ON experiment. KSM-ON (best)

Tab. 1: Impact of memory de-duplication on Redis tail latency

	set requests		get requests	
	p50	p99	p50	p99
Linux	0.5 ms	9.7 ms	0.5 ms	9.3 ms
Linux+KSM	0.5 ms	20.9 ms	0.5 ms	20.8 ms
CheriBSD	2.1 ms	3.7 ms	2.2 ms	3.6 ms
CheriBSD+CC	2.1 ms	4.2 ms	2.1 ms	4.3 ms
CheriBSD+CC+ORC	2.1 ms	4.9 ms	2.1 ms	4.8 ms

peaks faster than KSM-tuned, but instances are created more slowly and peak at just 140 (98.6% of ORC) because KSM is constantly consuming a lot more CPU resources. KSM-ON (worst) creates instances at the same rate as KSM-ON (best), until its probabilistic heuristics stop at only 100 instances (70% of ORC).

Conclusions: The results show that ORC is much more efficient than KSM at de-duplicating memory, as long as it is explicitly identified. Furthermore, an aggressive KSM configuration such as KSM-ON wastes CPU resources and drives aggregate application throughput as low as 70% of ORC, whereas a more conservative and adaptive configuration such as KSM-tuned wastes memory resources and takes up to 20× longer to de-duplicate memory, making the system ineffective in scenarios with frequent process creation and destruction.

Large binary instances. We now evaluate how a larger amount of shareable memory affects ORC and KSM, given that language runtimes and programming frameworks can easily consume hundreds of megabytes[1]. To this end, we run the same experiment after manually injecting an additional 100 MB of code on the FFmpeg binary, resulting in 53% of shareable contents on each identical instance.

Fig. 5 shows the results for this experiment, which supports up to 68 and 136 instances without and with perfect code de-duplication, respectively. We see the same expected pattern when looking at the KSM-tuned and ORC results, where KSM is further disadvantaged due to its runtime performance overheads (136 instances in 18 secs vs. 377 secs). In this case, we also report the best results for KSM-ON with two fixed values for `pages_to_scan` (1,200 and 20,000), which reach 132 instances 52 secs earlier than the default policy.

Conclusions: As we increase the amount of shareable memory, ORC remains optimally effective and becomes comparatively faster than KSM at de-duplicating (20× vs. 21×).

5.3 Impact on service tail latency

In this experiment, we investigate the impact of ORC and KSM on the tail latency of a typical cloud service. We observe that KSM consumes up to an entire CPU core when active, and

[1]For example: database system MongoDB: 104 MB, machine learning stack LibTorch with CUDA: >200 MB, Python-based data science pipeline with TensorFlow: >300 MB

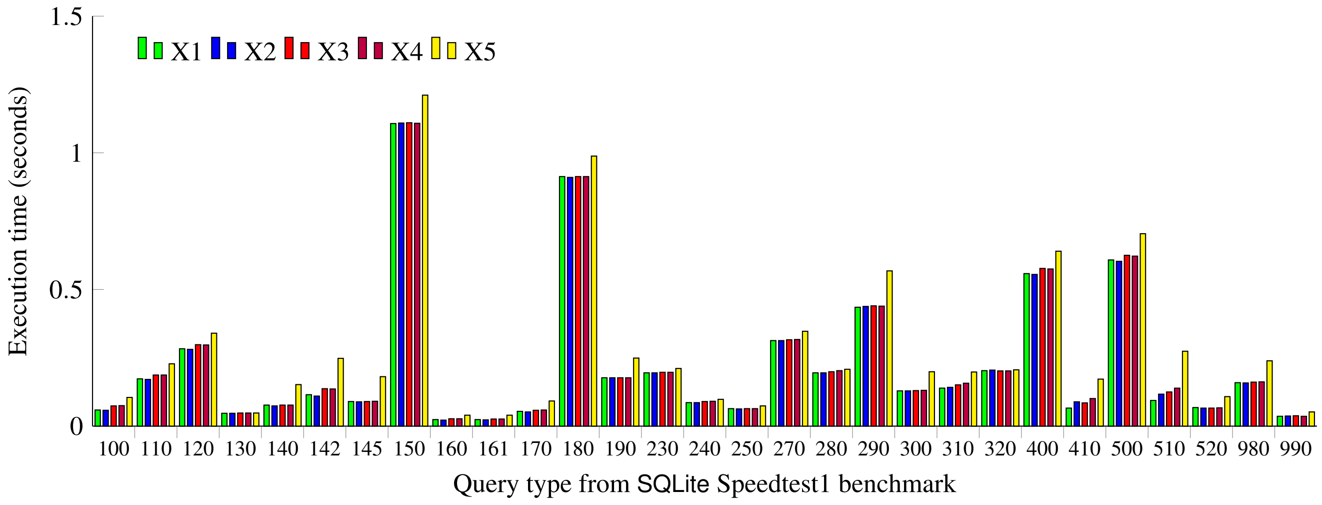


Fig. 6: Query execution times for different compartments using ORC (SQLite)

can issue TLB shootdowns when scanning and de-duplicating pages.

We spawn four instances of the Redis key-value store service [48] and use the *memtier* benchmark [37] as a workload: it pre-fills each instance with 2.5G GB of data and then executes a 1:10 set:get request workload, using 4 threads with 4 concurrent connections for each instance.

Tab. 1 shows the results of four different deployments: Linux acts as our baseline (arm64 binaries without using CHERI or KSM); Linux+KSM adds memory de-duplication with an always-on KSM; CheriBSD is our baseline with a CHERI-capable host OS but without ORC or enabling capabilities when compiling Redis; CheriBSD+CC uses ORC to compartmentalize Redis but disables de-duplication; and CheriBSD+CC+ORC uses ORC for memory de-duplication.

Note that we run both Linux and CheriBSD to decouple the impact of KSM and ORC from the intrinsic differences between the two host OSs. CheriBSD shows worse throughput and mean/tail latencies than Linux on all operations, which can be attributed to the different device driver and network stack implementations.

Linux+KSM retains the p50 latencies of Linux, but the CPU overheads and TLB shootdowns introduced by KSM more than double the p99 latency. In contrast, ORC has a very small impact on tail latency. Support for compartmentalization alone (CheriBSD+CC; i.e., compiling the program in pure-cap mode and crossing isolation boundaries) drives a 13% and 19% increase in p99 latency for set and get operations, respectively. Fully enabling ORC (CheriBSD+CC+ORC) has an additional 17% and 12% increase in p99 latency, can be attributed to the overheads of our current implementation for CLS accesses.

Conclusions: The necessary page table management and memory hashing overheads of blind page de-duplication of

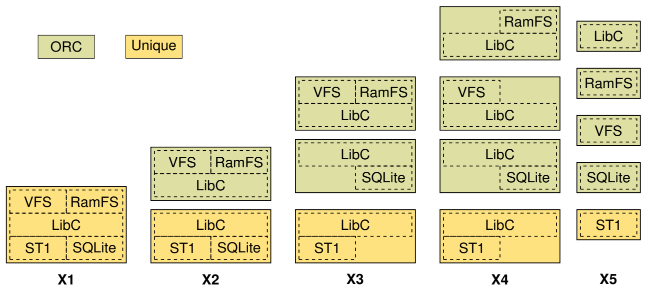


Fig. 7: Decomposition into capability compartments (SQLite)

KSM leads to more than 2× increase in p99 latency, whereas the explicit sharing of binary objects in ORC limits these overheads to 12–17%.

5.4 Cost of isolation

Finally, we examine how increasing the number of shareable binary objects in an application affects its performance. We control both the overheads introduced by our CLS compiler pass, and those of capability-based crossings between binary objects.

To measure this, we execute a single compartment with the *speedtest1* benchmark and its embedded SQLite instance, and incrementally build some of its components into different binary objects, as shown in Fig. 7. We start with X1, which statically contains all components into a single binary object, where we disable the compiler pass for CLS (we do not need CLS with a single object and compartment). We then incrementally build some of these components into separate shareable objects in X2–X5, where each additional object is compiled with CLS support. The *VFS* and *RamFS* components correspond to internal components of our library OS, whereas *ST1* corresponds to the benchmark binary.

Fig. 6 shows the execution times of the Speedtest1 benchmark for different SQLite query types and each of the binary object configurations. We can see that all configurations, except X5, have either a minor or negligible performance overhead. The average difference between X1 (everything monolithic; no CLS support) and X4 (all system components but LibC are shareable objects) is 10%, with a median of 3%. Even when isolating all the various components into separate shareable objects (X5), which suffers from many cross-object calls, has an average slowdown of 53%, with a median of 39%.

Conclusions: Based on the results, we can draw two conclusions. First, the overhead of supporting CLS is very small, especially when compared to the performance cost of cross-object calls. Second, while the cost of cross-object calls exists, it is small enough that we can afford sharing across multiple fine-grained objects.

6 Related Work

Page-level memory sharing. Several modern hypervisors support dynamic page sharing among VMs. VMWare ESX [61] pioneered inter-VM page sharing without guest OS support by periodically scanning physical pages and transparently discovering pages with identical contents using their hash values. KSM [1] also periodically scans physical pages but uses a balanced tree to find duplicated pages (see §2.1). Dynamic page sharing by hypervisors, however, results in an inflexible sharing granularity due to the lack of OS semantics, unpredictable latency spikes due to runtime scans, and vulnerabilities to side-channel attacks due to copy-on-write semantics.

Hypervisors that perform page sharing on disk reads have also been proposed. Disco [5] intervenes in DMA to support copy-on-write shared disks and copy-less NFS shares among VMs, allowing page sharing without runtime scanning. Satori [41] also proposed similar sharing-aware block devices that enable copy-on-write sharing as well as content-based sharing through enlightenment (para-virtualization). Sharing in these systems is still page-based, and copy-on-write issues also remain.

VM introspection (VMI) or graybox approaches can be used to improve the efficiency of deduplication by extracting semantic information from in-VM memory data. Sindelar et al. [53] used VMI techniques to identify memory pages belonging to free memory pools in Windows and Linux without making kernel version-specific assumptions, and treated them as zero pages to improve deduplication and VM migration efficiency. Singleton [52] uses KSM page information to deduplicate pages in the guest page cache by dropping them from the host page cache. VMI, however, is not always able to obtain semantic information reliably without cooperation from the guest, and these techniques are still page-based.

Overall, ORC has the advantage over page-level sharing in that it allows for reliable, flexible, and efficient sharing that leverages the semantic information about objects.

Efficient inter-VM page sharing techniques can be applied to optimize inter-server VM placement for cloud-wide memory density. *Memory buddies* [65] aggregates memory fingerprint information into a centralized control plane to determine VM placement for increased sharing and dynamically optimizes VM placement with live migration. Sindelar et al. [53] show that inter-VM sharing largely occurs hierarchically and use a tree structure to manage sharing. ORC could leverage semantic knowledge about shared memory objects and could be applied to improve memory density in the cloud through optimal VM placement.

Mixed-granularity sharing. Sharing at a finer granularity than pages can help increase memory density, as many pages are found to be nearly identical with only some differences. Gupta et al. [28] propose a *Difference Engine* as an extension to Xen [2], which supports sharing at the sub-page level in addition to page level. The Difference Engine stores patches against reference pages for similar but not identical pages, and compresses pages that are unique but accessed infrequently. Several studies on VM live migration also leverage sub-page granularity for differentiation, compression and write detection [15, 46, 71]. Although such proposals could increase memory density, unlike ORC, they do not semantically reason about what should be shared across different tenants.

Several studies to improve scanning efficiency by grouping pages based on access characteristics use a granularity finer than pages. CMD [8] identifies page access characteristics by measuring the distribution of writes per subpage within a page using dedicated hardware, in addition to the address and number of writes to the page. UKSM [68] proposes adaptive partial hashing, which hashes only a portion of the page and gradually changes its size. These studies allow for fine-grained sharing and reduce the cost of hash computation, but still lack semantic information, making sharing opportunities non-deterministic.

In recent virtualized environments, it has become increasingly important to leverage large page sizes to minimize the overhead of TLB misses, while the opportunity for page-level memory sharing decreases as the page size increases. SmartMD [27] splits large but cold pages with high repetition rates for de-duplication, while reconsolidating small but hot pages for improved memory access performance. GLUE [47] attempts to maintain large-page performance in regions that are broken into small pages (splintered) for de-duplication by extending the hardware to perform speculative large-page translation while using normal-sized TLBs. These studies mitigate inefficiencies caused further by large pages, but page-level issues still remain.

Overall, while these studies attempt to leverage finer granularity than a page, ORC still has a significant advantage in its ability to share per-object at byte granularity using semantic information.

7 Conclusions

We have described ORC, a new memory de-duplication approach that improves the memory density of cloud environments using capability-protected compartments. Our motivation was to create a practical, capability-based cloud stack, in which tenants can enjoy strong isolation and cloud providers benefit from more efficient use of memory resources.

Unlike conventional hypervisors, which blindly scan memory for identical memory pages, ORC takes advantage of a semantic separation of sharable and non-sharable objects. Therefore, ORC is not subject to the performance overhead of existing runtime methods that arise from the scanning and deduplication of pages. Thanks to its use of capabilities, ORC allows for more precise sharing of memory objects at a word granularity (i.e., spatial precision), while avoiding unintentional sharing of runtime objects (i.e., temporal precision). The management of compartments is done via a narrow interface with a small TCB, providing strong isolation guarantees.

Discussion. In this paper, we use a capability-based architecture to increase memory density. The proposed design relies solely on hardware memory capabilities. However, some design decisions could be implemented differently by using a hybrid approach that is based on the combination of MMU and capabilities, or only MMU, due to similarities in functionality that exist between these technologies. For example, sharing of code between cVMs and sharing of pages between processes/VMs are similar. However, architectural capabilities offer additional mechanisms and security features compared to that of MMU, such as spatial memory protection and fast switches between compartments. To avoid duplication of features, we are considering future systems as no-MMU, where protection and isolation are wholly implemented at the ISA level. As a consequence, our solution is entirely based on capabilities and applicable to both MMU-based and no-MMU architectures.

Acknowledgements. This work was funded by the Technology Innovation Institute (TII) through its Secure Systems Research Center (SSRC), and the UK Government's Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme (UKRI grant EP/V000365 "CloudCAP"). This work was also supported by JSPS KAKENHI Grant Number 18KK0310 and JST CREST Grant Number JPMJCR22M3, Japan.

Source code availability. The source code of ORC, the LLVM pass, and various application examples can be found at https://github.com/lsds/intravisor.

References

[1] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium 2009*, pages 19–28, 2009.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177. ACM, 2003.

[3] Luiz André Barroso, Jimmy Clidaras, and Urs Hlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2nd edition, 2013.

[4] Viktors Berstis. Security and protection of data in the IBM System/38. In *Proceedings of the 7th annual symposium on Computer Architecture*, ISCA '80, pages 245–252, New York, NY, USA, May 1980. Association for Computing Machinery.

[5] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[6] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-Based Addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 319–327, New York, NY, USA, 1994. Association for Computing Machinery.

[7] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, pages 244–249. IEEE, 2011.

[8] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. CMD: Classification-based Memory Deduplication through Page Access Characteristics. *ACM SIGPLAN Notices*, 49(7):65–76, 2014.

[9] TIS Committee et al. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2, 1995.

[10] Intel Corporation. 5-level paging and 5-level EPT. Technical report, Intel Corporation, May 2017. Revision 1.1.

[11] Domenico Cotroneo, Roberto Natella, and Roberto Pietrantuono. Predicting aging-related bugs using software complexity metrics. *Performance Evaluation*, 70(3):163–178, 2013. Publisher: Elsevier.

[12] CVE-2013-6441. Available from MITRE, CVE-ID CVE-2013-6441, December 2013.

[13] CVE-2021-21284. Available from MITRE, CVE-ID CVE-2021-21284, December 2021.

[14] Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[15] Umesh Deshpande, Xiaoshuang Wang, and Kartik Gopalan. Live Gang Migration of Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, page 135–146, 2011.

[16] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008.

[17] Ulrich Drepper. *ELF Handling For Thread-Local Storage*, August 2013. Version 0.21.

[18] DM England. Capability concept mechanism and structure in System 250. In *Proceedings of the International Workshop on Protection in Operating Systems*, pages 63–82, 1974.

[19] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[20] Wei Fan and Albert Bifet. Mining big data: Current status, and forecast to the future. *ACM SIGKDD Wxplorations Newsletter*, 14(2), 2013.

[21] FFmpeg: A complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org. 2022.

[22] FreeBSD adapted for CHERI-MIPS, CHERI-RISC-V, and Arm Morello. https://github.com/CTSRD-CHERI/cheribsd. Last accessed: June 1, 2022.

[23] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing os abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[24] Robert A Gingell, Meng Lee, Xuong T Dang, and Mary S Weeks. Shared libraries in sunos. *AUUGN*, 8(5):112, 1987.

[25] Mel Gorman. *Understanding The Linux Virtual Memory Manager*. Prentice Hall Upper Saddle River, 2004.

[26] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed Javascript. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, volume 9326, pages 108–122. Springer International Publishing, Cham, 2015. Series Title: Lecture Notes in Computer Science.

[27] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John CS Lui. SmartMD: A High Performance Deduplication Engine with Mixed Pages. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 733–744, 2017.

[28] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 309–322, December 2008.

[29] Gernot Heiser. The seL4 microkernel – an introduction, June 2020. White paper. The seL4 Foundation, Revision 1.2 of 2020-06-10.

[30] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[31] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. Efficient memory sharing in the xen virtual machine monitor. *Aalborg University*, pages 1–86, 2006.

[32] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 376–394, 2021.

[33] Sajib Kundu, Raju Rangaswami, Ming Zhao, Ajay Gulati, and Kaushik Dutta. Revenue Driven Resource Allocation for Virtualized Data Centers. In *2015 IEEE International Conference on Autonomic Computing*, pages 197–206, July 2015.

[34] Lanfranco Lopriore. Capability based tagged architectures. *IEEE transactions on computers*, 33(09):786–803, 1984.

[35] Linux containers. https://linuxcontainers.org. Last accessed: June 1, 2022.

[36] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *Intl. Symp. on Microarchitecture (MICRO)*, 2019.

[37] NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark. Last accessed: Dec 13, 2022.

[38] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2, 2014.

[39] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 279–290, 2013.

[40] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 279–290, 2013.

[41] Grzegorz Milos, Derek G Murray, Steven Hand, and Michael A Fetterman. Satori: Enlightened page sharing. In *USENIX Annual technical conference*, 2009.

[42] Arm Morello Program. https://www.arm.com/architecture/cpu/morello. 2022.

[43] musl libc. https://musl.libc.org. Last accessed: June 1, 2022.

[44] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[45] Rodney Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *30th IEEE International Performance Computing and Communications Conference*, pages 1–8, November 2011. ISSN: 2374-9628.

[46] Yosuke Ozawa and Takahiro Shinagawa. Exploiting Sub-page Write Protection for VM Live Migration. In *Proceedings of the 2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 484–490, 2021.

[47] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: can you have it both ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 1–12, Waikiki Hawaii, December 2015. ACM.

[48] Redis is an in-memory database that persists on disk. https://github.com/redis/redis. Last accessed: June 1, 2022.

[49] Vasily A. Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. CAP-VMs: Capability-Based isolation and sharing in the cloud. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 597–612, Carlsbad, CA, July 2022. USENIX Association.

[50] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 575–587. ACM, 2021.

[51] Martin Schwidefsky, Hubertus Franke, Ray Mansell, Himanshu Raj, Damian Osisek, and JongHyuk Choi. Collaborative memory management in hosted linux environments. In *Proceedings of the Linux Symposium*, volume 2, pages 313–330. Linux Symposium Incorporation Ottawa, 2006.

[52] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing - HPDC '12*, page 15, Delft, The Netherlands, 2012. ACM Press.

[53] Michael Sindelar, Ramesh K. Sitaraman, and Prashant Shenoy. Sharing-aware algorithms for virtual machine colocation. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures - SPAA '11*, page 367, San Jose, California, USA, 2011. ACM Press.

[54] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. Pageforge: a near-memory content-aware page-merging architecture. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 302–314, 2017.

[55] Speedtest1 benchmark. http://www.sqlite.org/src/finfo?name=test/speedtest1.c. Last accessed: Dec 13, 2022.

[56] Sqlite. https://www.sqlite.org. 2022.

[57] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the Fifth European Conference on Computer Systems*, EuroSys '10, pages 209–222. ACM, 2010.

[58] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Software side channel attack on memory deduplication. In *ACM Symposium on Operating Systems Principles (SOSP 2011), Poster session*, 2011.

[59] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Implementation of a Memory Disclosure Attack on Memory Deduplication of Virtual Machines. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E96.A(1):215–224, 2013.

[60] A Virtualization. Secure virtual machine architecture reference manual. *AMD Publication*, 33047, 2005.

[61] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2003)*, December 2003.

[62] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W Moore, et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical report, University of Cambridge, Computer Laboratory, 2019.

[63] Robert NM Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W Moore, Edward Napierala, Peter Sewell, et al. CHERI C/C++ Programming Guide. Technical report, University of Cambridge, Computer Laboratory, 2020.

[64] Maurice Vincent Wilkes and Roger Michael Needham. The Cambridge CAP computer and its operating system. *Operating and Programming System Series*, 1979.

[65] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. *ACM SIGOPS Operating Systems Review*, 43(3):27–36, 2009. Publisher: ACM New York, NY, USA.

[66] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[67] Lei Xia and Peter A. Dinda. A case for tracking and exploiting inter-node and intra-node memory content sharing in virtualized large-scale parallel systems. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date - VTDC '12*, page 11, Delft, The Netherlands, 2012. ACM Press.

[68] Nai Xia, Chen Tian, Yan Luo, Hang Liu, and Xiaoliang Wang. Uksm: Swift memory deduplication via hierarchical and adaptive memory region distilling. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, pages 325–339, February 2018.

[69] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, Budapest, Hungary, June 2013. IEEE.

[70] Bernd Zeimetz. ksmtuned. `https://github.com/bzed/debian-ksmtuned`. 2022.

[71] Xiang Zhang, Zhigang Huo, Jie Ma, and Dan Meng. Exploiting Data Deduplication to Accelerate Live Virtual Machine Migration. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, pages 88–96, 2010.