ORC: Increasing Cloud Memory Density via Object Reuse with Capabilities

Vasily A. Sartakov, Lluís Vilanova, and Munir Geden, *Imperial College London;* David Eyers, *University of Otago;* Takahiro Shinagawa, *The University of Tokyo;* Peter Pietzuch, *Imperial College London*

https://www.usenix.org/conference/osdi23/presentation/sartakov

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

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 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 ORC achieves a higher memory density with a lower overhead than hypervisor-based de-deduplication.

1 Introduction

In data centers, memory density determines how many applications can be deployed on machines with given memory amounts. Therefore, density is a critical cost factor, as memory contributes significantly to capital and operational expenses [3]. The challenge of achieving high memory density is expected to worsen as applications move to larger working set sizes [20, 36], and machines have more memory [10].

High memory density can be achieved by *de-duplicating* memory pages that have the same contents across a 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 level of isolation between tenants. Containers and processes achieve near-perfect memory density when they use a shared OS with 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*. For security reasons, 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 OS. While this removes complexity from hypervisors, allowing them to provide strong isolation, it loses semantic information about how memory pages are used by VMs for 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 page scanning [1, 31, 40, 51], this does not eliminate overheads.

Our goal is to design a new cloud software stack that combines high memory density with low overhead, while providing strong isolation guarantees between tenants, relying only on a small trusted computing base (TCB).

We describe **Object Reuse with Capabilities (ORC)**, a new cloud software stack that allows de-duplication across tenants with strong isolation, a small TCB, and low overheads. ORC extends a binary program format (ELF [9]) to enable isolation domains to share binary objects, i.e., programs and libraries, by design. Object sharing is always explicit, thus avoiding the performance overheads of hypervisors with page de-duplication. For strong isolation, ORC only shares immutable and integrity-protected objects. To keep the TCB small, object loading is performed by the untrusted guest OS.

In more detail, the design and implementation of ORC combines the following novel features:

(1) Object sharing with capabilities. Current cloud stacks use 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 latencies [54].

Instead, ORC shares binary objects by design between isolation domains: it uses hardware support for *memory capabilities* [14, 16, 34, 64, 66] to place objects into *compartments*. Memory capabilities grant access to memory regions, can be copied between memory and registers, and are protected by hardware. They can 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].

With the help of capabilities, ORC isolates multiple compartments within a single address space, while sharing binary objects between compartments with virtually no overhead. ORC uses memory capabilities to isolate compartments within a single page table, safely and efficiently sharing objects in a controlled way.

(2) Safe sharing of immutable objects. 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. While this is not an issue with per-process page tables, because an object's global variables are mapped to different physical addresses in each process, ORC's shared page table means that global per-process-and-object variables must be handled differently when sharing pages 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 in 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. The complexity of object loading 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 (i.e., storage and file system I/O, data processing and copying, and adjustments of memory contents). When an object is requested for the first time, the untrusted compartment manages its loading, and requests ORC to register an immutable and integrity-protected version of the newly-loaded object.

ORC verifies that the loaded object cannot be used to attack future compartments that reuse the same object, and makes it available to future load requests. The verification process is simple: it requires (i) scanning the memory contents of the registered object to calculate a hash, which ensures the object's integrity in future load requests; and (ii) checking that any contained capabilities used by the object stay within the object's memory and maintain immutability.

We evaluate ORC using a prototype implementation on the CHERI/Morello platform [42, 66] that includes a new compiler pass and loader support for CLS, a small privileged component that manages compartments and enforces the properties for secure sharing of binary objects, and a port of a library OS and C standard library that execute in each compartment.

We consider three workloads: (1) a cloud-based video transcoding micro-service, showing that ORC's memory de-duplication is more resource-efficient compared to page-level de-duplication; (2) a latency-sensitive key/value store, demonstrating the lower impact of object-level de-duplication on tail latencies; and (3) an embedded database system, evaluating ORC's decomposition cost into sharable compartments.

ORC has several limitations: unlike hypervisor-based de-duplication, ORC only de-duplicates read-only contents; it needs applications recompiled to use capability instructions and the new CLS; and it executes all compartments in one virtual address space, because capabilities use virtual addresses.

2 Increasing Memory Density in the Cloud

Memory density in cloud computing defines how efficiently the cloud provider is utilizing 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 from those of other tenants, while providers seek ways to minimize the total physical memory footprint by finding shareable memory. The two goals are at odds with each other: the

mechanisms that we have for efficient memory sharing are, in essence, reducing the level of isolation between tenants.

With today's isolation approaches, we must choose between containers [35, 38] and VMs [2, 30, 61], which in turn dictate how memory sharing can be done:

(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 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 extra runtime overhead.

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

(2) Hypervisor-managed memory sharing. VMs offer stronger isolation compared to containers: they expose 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. Since the hypervisor lacks visibility into the semantics of user-space applications within each VM, 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 KVM hypervisor leverages to eliminate duplicate memory pages across VMs. KSM periodically scans physical memory to find identical pages, and deduplicates them by mapping a single physical copy to multiple virtual locations. It also marks those pages as copy-on-write (COW), which triggers the re-duplication before a modification on shared page contents. Therefore, each VM instance can safely operate on its own copy, without affecting the memory contents of other VMs.

KSM uses red-black trees to search for memory pages with identical content. For efficiency, it utilizes two 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, 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 hashes. 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.

Despite the advantages of hypervisor-based isolation, its memory de-duplication mechanism has several drawbacks: (1) blindly scanning page contents and manipulating their permissions comes with an overhead on average performance and tail latencies [8, 39, 54]; (2) hypervisors lack the visibility of an OS to application-level load-time semantics, and therefore must rely on page scans; (3) while the use of larger pages in the cloud improves memory performance [27, 47], it can reduce memory density by making memory de-duplication less frequent; and (4) the use of COW semantics 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

As described above, using paging for both translation and protection introduces performance challenges in traditional virtualized environments due to its management granularity. 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 regular pointers, capabilities provide information to enforce accesses within a given address range and access type. They can thus be used to partition a single address space into multiple, isolated regions, allowing the use of a single page table across isolation domains.

Memory capabilities. The Capability Hardware Enhanced RISC Instructions (CHERI) architecture [62] 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, but can only be derived from another capability; (2) *capability integrity* guarantees that capabilities in memory cannot be modified. One-bit *validity tags* are used to distinguish them from other data for protection; 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 enforce precise bounds and permissions on each memory access. This is known as the *pure-capability* (*pure-cap*) mode. 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 supports a *hybrid* mode, in which code is permitted to use legacy, capability-unaware instructions. In this mode, all control flow and memory operations are checked against a pair of special registers that contain *program-counter* and *default data capabilities*, thus restricting the code and data accesses performed by capability-unaware instructions.

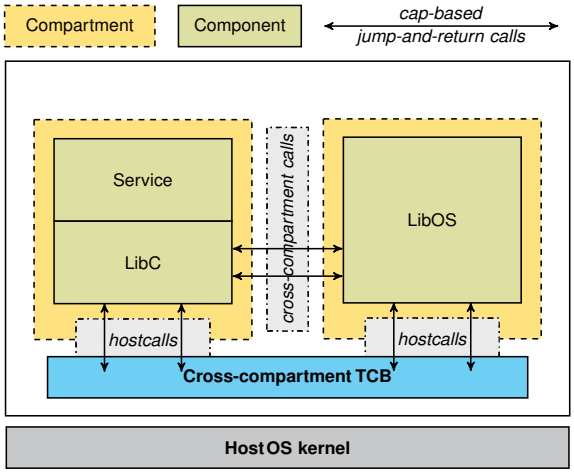


Fig. 1: Using capabilities to create compartments

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

Capability-based compartmentalization. Capabilities offer a good mechanism to isolate software components, and their flexibility and efficiency can eliminate the overheads of page-based sharing. CAP-VM [49] describes a new capability-based compartmentalization mechanism: each capability compartment (cVM) has its own separate address sub-range within a shared address space, and executes programs in CHERI's 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 hybrid mode to isolate cVMs. Each cVM has its own library OS instance to support private namespaces and program execution environments.

Fig. 1 shows how multiple components can be placed in capability compartments, potentially sharing access to objects across compartments. An Intravisor, or some other cross-compartment TCB, can give each compartment the capabilities needed to jump into/call code in other compartments, allowing the compartments to share capabilities to the same object. The figure also shows how capabilities can be used to request Intravisor operations (*hostcalls* at the bottom).

After compiling software components using CHERI's pure-cap mode, however, it is not possible to use capability-based compartments to share objects efficiently, because: (1) the Intravisor has no explicit information about the extent and sharing properties 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, ELF [9] assumes 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 page table management overheads.

2.3 Efficiency and security considerations

The goal of this paper is to provide high 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 optimizations, 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.

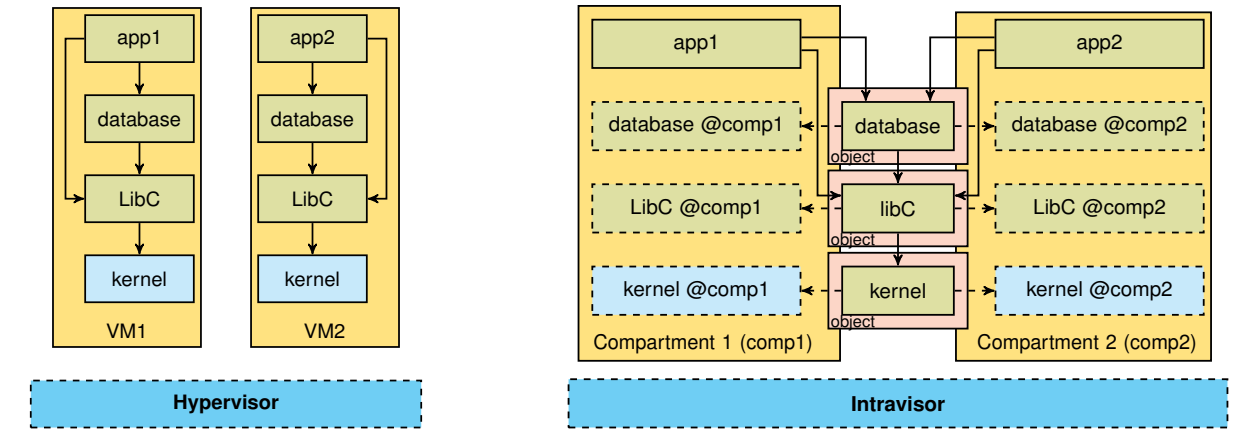
Note that sharing memory can be used as an unintended *side channel* across compartments. This is an intrinsic trade-off between memory density and isolation that all cloud providers face, regardless of the employed mechanism. Given the importance of side channels, there are proposals to avoid or mitigate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

We exploit the capabilities provided by the CHERI architecture [62] to implement both software compartmentalization and binary object sharing. ORC shares binary objects *explicitly*, making the use of physical memory denser without the performance overheads of de-duplication.

Fig. 2 shows an example with two applications (app1 and app2), which use multiple object binaries that are identical across VMs, including the OS kernel (database, libC and kernel). Fig. 2a shows a baseline system in which each application is deployed in a VM for maximum isolation. In this case, the hypervisor incurs overheads of memory de-duplication.



(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.)

In contrast, Fig. 2b shows the approach taken by ORC. Programs are isolated into *compartments* (shown as light yellow boxes), which contain all 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 they 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, allowing future compartments to 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. In addition, sharing is part of the cloud software stack, ensuring that the memory density benefits do not come at the cost of reduced performance.

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 binary, 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, by extending the binary storage format and the loader (see below). The figure shows an example with three global variables, a constant, a thread-local, and 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 use private copies of objects without the ORC extensions, and only references to global writable variables are changed to the CLS. Heap allocations are supported as usual, resulting in private compartment allocations.

Each compartment has its own, untrusted loader (`ld.so` in Linux). The compartment itself therefore loads the required objects by reading them from storage and parses 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 and prepares it for execution.

❸ After loading, the compartment calls the Intravisor's `orc_register()` operation to 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 the 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

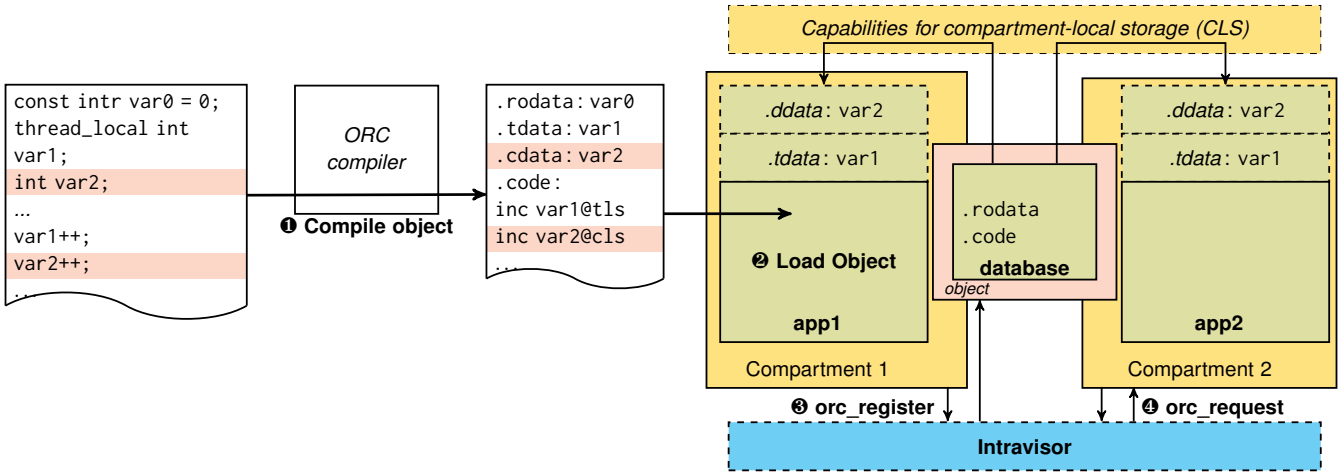


Fig. 3: Architecture of the ORC stack (Circled numbers identify operations referenced in the text.)

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 CLS variables. With ORC enabled, the compiler adds a flag to identify the object as ORC-enabled, and moves writable global variables to the CLS.

For CLS, the compiler replaces each reference to a writable global 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.

Fig. 3 shows this process on the left-hand side. Code, constant variables, such as `var0`, and thread-local variables, such as `var1`, are handled as usual by the compiler: they are placed in the `.code`, `.rodata` and `.tdata` sections of the ELF object, respectively, generated with standard relocations. Writable global variables (`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 malicious use of `orc_register()`. At this point, the Intravisor computes a hash H of the object contents, 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 §4.2).

When a compartment calls `orc_request()`, it passes the hash H. If an object with hash H exists in the Intravisor, i.e., 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 that 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 the software supply chain, for which hash H can be helpful.

3.4 Discussion

With ORC, components are shared by design via capabilities. Component sharing by design could also be implemented by other systems, e.g., traditional hypervisors with MMU mappings, but end-to-end performance would vary due to the different hardware mechanisms for enforcing isolation. Capabilities have further benefits e.g., their ability to provide spatial memory protection within components.

We also note that ORC only de-duplicates read-only mem-

ory contents, while traditional hypervisors also de-duplicate writable pages. This is not necessarily an issue: ORC de-duplicates application instances in less time than hypervisor-based approaches, which makes it better suited for short-lived instances, as often found in cloud environments. Our evaluation results also show that de-duplicating small amounts of writable memory leads to little practical benefit.

Finally, ORC requires the recompilation of applications, because applications must use capability instructions for code and data access and ORC's compiler pass for the required CLS functionality. ORC also executes all compartments in a single virtual address space, but we expect this to not be a problem: virtual addresses are an abundant resource, and it is possible to use large blocks to make allocations scalable and resistant to side channels e.g., via core-local caches.

4 Implementation

We implement ORC on the Morello platform [42], a development board from Arm that supports 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 load shared objects securely.

Both the Intravisor and the host OS are implemented as hybrid capability code using the existing CAP-VM 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, the main effort went into porting code to the pure-cap model.

4.1 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.

4.2 Compiler support and CLS

We implement our ORC compiler support as an LLVM pass. It replaces all references to global, writable, non-TLS variables with a call to function `__cls_get_addr()`, which returns a 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 ELF [17]. It takes the shared object identifier (assigned at load time) and the variable offset within the CLS (assigned at compile time), and returns a capability that grants access to the calling compartment's copy of that variable.

For ease of implementation, the 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 (see below) only needs to implement TLS variables, accessed through the `tp` register in Arm, to support both TLS and CLS.

4.3 Secure object loader

To simplify application deployment, we implement a loader that takes deployment configurations, i.e., a 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 configuration also identifies shareable objects and provides their hash, so that they can be reused if previously loaded by `orc_register()` and `orc_request()`.

4.4 Discussion

Our prototype implementation showcases ORC's core ideas, but it has shortcomings:

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

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.

Our `__cls_get_addr()` implementation assumes a per-compartment CLS. We plan to support per-process CLS, allowing for multiple processes within the same compartment.

5 Evaluation

We ask the following questions when evaluating ORC: (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 setup

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 of ORC and KSM on request tail latencies; and (2) we use the SQLite database system [56] and its *speedtest1* benchmark [55] to compare the performance of different object sharing scenarios.

Testbed. 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 run ORC with all its isolation guarantees, as described in this paper. All ORC results use CheriBSD unless stated otherwise.

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.

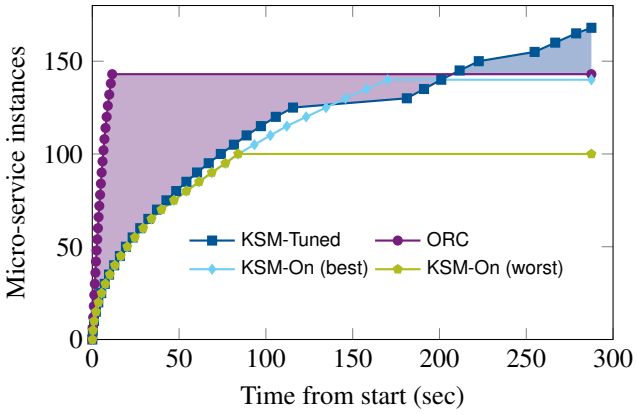


Fig. 4: Throughput over time when de-duplication of the video transcoding micro-service happens (The shaded area indicates the difference between ORC and KSM-Tuned, which is the best-performing KSM configuration.)

5.2 Efficiency and performance overhead

We now evaluate the trade-off between memory de-duplication efficiency and application performance overhead when comparing ORC and KSM. We deploy the video transcoding micro-service, which increases application throughput with higher memory density – i.e., it increases the number of processed frames by increasing the number of deployed transcoder instances.

The experiment deploys new transcoder instances until it reaches 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 contains FFmpeg's main program and libraries, our library OS, and the standard C library (see §4.1). It occupies around 111 MB of memory (11 MB in binaries and read-only data, shareable 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 reaches the memory limit after 127 instances; optimal de-duplication can run more instances: it eventually reaches the CPU limit after 180 instances.

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

KSM-Tuned 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 adjusts KSM parameters: the number of scanned pages in each iteration (`pages_to_scan`) is increased gradually when the de-duplication rate is too low.

KSM-On is a hand-tuned policy that achieves good memory efficiency in the shortest time for our workload, but it consumes significant CPU resources. With this setting, KSM operates constantly and uses 20,000 `pages_to_scan`.

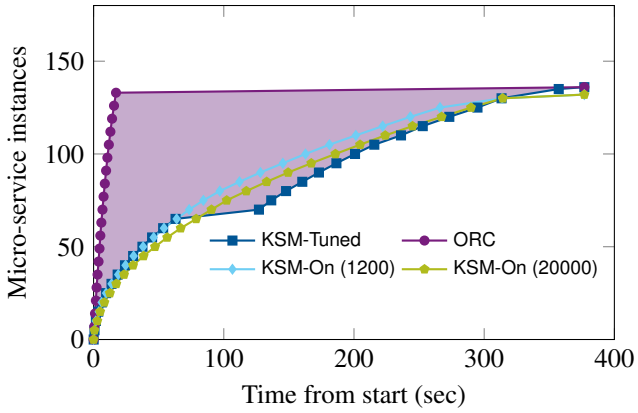


Fig. 5: Throughput over time when de-duplication of the video transcoding micro-service happens using larger binaries (The shaded area indicates the difference between ORC and KSM-Tuned.)

Micro-service instances. Fig. 4 shows the number of active micro-service instances over time, with ORC and the various KSM configurations. Given the performance of one instance, there are sufficient CPU resources for 180 instances, but only enough memory for 127 instances without de-duplication.

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

In contrast, ORC is designed to optimize for density almost instantaneously: it creates 142 instances in just 11 secs, which 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 without de-duplication. This is expected, because code and read-only data occupy only 11% of the memory of each instance (see above). In contrast, KSM can de-duplicate additional pages by also considering writable memory.

ORC thus deploys and de-duplicates instances much faster than KSM-Tuned, which gives it an advantage over KSM on the aggregate performance over time, expressed as the total number of processed frames (highlighted by the shaded areas in Fig. 4). Although KSM-Tuned deploys more instances than ORC within 300 secs, at this point, ORC has processed 15%–35% more frames than the various KSM configurations. To outperform ORC, KSM-Tuned would require a total execution time of 441 secs – an extra 141 secs after reaching 180 instances. None of the other policies outperform ORC.

KSM-Tuned limits its de-duplication speed, as it tries to minimize CPU overheads – it operates only under memory pressure (80%), and at a limited memory scanning rate (`pages_to_scan`). This negatively impacts environments in which processes are frequently created and destroyed, so we also evaluate the case in which KSM maximizes de-duplication rate with KSM-On. Since KSM is probabilistic in nature, we show the best and worst results of twenty different runs of the same KSM-On experiment.

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

KSM-On (best) deploys more instances than KSM-Tuned within the first 170 secs, but instances are created more slowly and peak at just 140 (98.6% of ORC), because KSM is constantly consuming more CPU resources. KSM-On (worst) creates instances at the same rate as KSM-On (best), until its heuristics stop at only 100 instances (70% of ORC).

Conclusions: The results show that instance creation and de-duplication in ORC are substantially more efficient than in the baseline system with KSM. Although KSM-Tuned de-duplicates more (writable) memory, given enough time, ORC retains an advantage with shorter-lived application instances, which are prominent in the cloud. In addition, more aggressive de-duplication with KSM-On is not viable, because higher de-duplication rates are hidden by higher CPU overheads, resulting in a 30% throughput overhead compared to ORC.

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 into the FFmpeg binary, resulting in 53% of shareable code and read-only data contents on each instance.

Fig. 5 shows the results for this experiment, which supports 68 and 136 instances without and with perfect de-duplication, respectively. In this case, the ability to de-duplicate writable data in KSM has no long-term advantage over ORC, because KSM has further CPU overheads that prevent spawning additional instances; ORC reaches 136 instances in 18 secs, while it takes KSM-Tuned 377 secs to reach the same maximum. 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: With a larger proportion of directly shareable contents, ORC reaches optimal de-duplication effectiveness at a 20× faster rate than any of the KSM configurations.

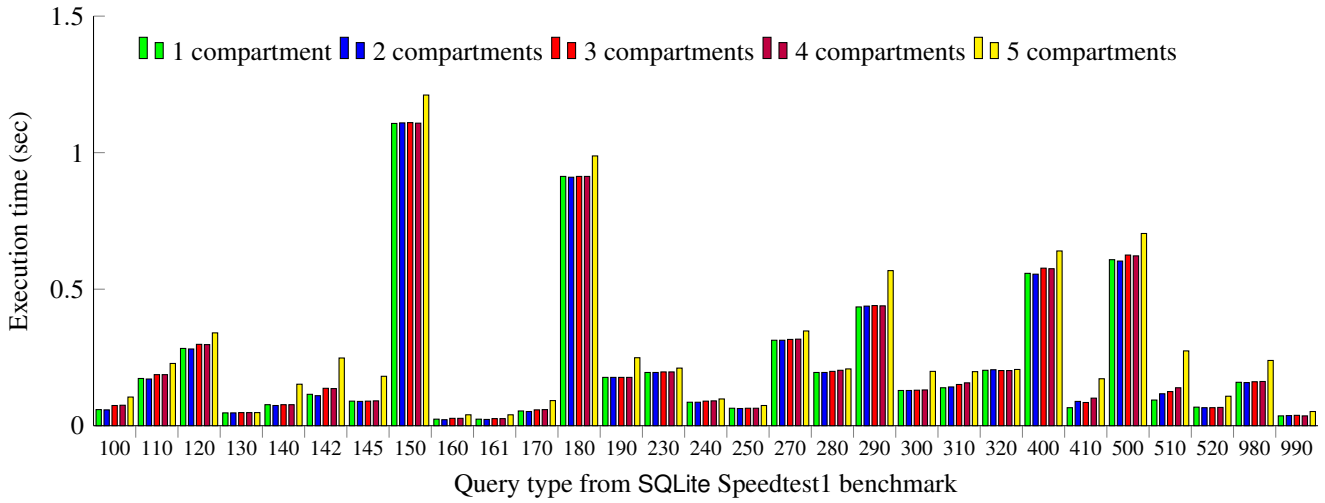


Fig. 6: Query execution times for different compartments using ORC (SQLite)

5.3 Impact on service tail latency

Next, we investigate the impact of ORC and KSM on the tail latencies in a typical cloud service. We observe that KSM consumes up to an entire CPU core when active, and it issues TLB shootdowns when scanning and de-duplicating pages.

We spawn 4 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 five deployments: (1) Linux acts as our baseline (arm64 binaries without using CHERI or KSM); (2) Linux+KSM adds memory de-duplication with an always-on KSM; (3) CheriBSD is our baseline with a CHERI-capable host OS but without ORC or enabling capabilities when compiling Redis; (4) CheriBSD+CC uses ORC to compartmentalize Redis but disables de-duplication; and (5) CheriBSD+CC+ORC uses ORC for de-duplication.

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

Linux+KSM matches the 50th percentile (p50) latencies of Linux, but the CPU overheads and TLB shootdowns due to KSM more than double the 99th percentile (p99) latencies. In contrast, ORC has a small impact on tail latencies: support for compartmentalization alone (CheriBSD+CC, i.e., compiling the program in pure-cap mode and crossing isolation boundaries results in a 13% and 19% increase in p99 latencies for set and get operations, respectively. Fully enabling ORC (CheriBSD+CC+ORC) leads to an 17% and 12% increase in p99 set/get latencies, respectively, which can be attributed to the overheads of our current CLS implementation.

Conclusions: The page table management and memory hashing overheads of KSM's page de-duplication lead to a more than 2× increase in p99 latencies; the explicit sharing of binary objects in ORC reduces these overheads to 12%–17%.

[1]For example, the MongoDB database system uses 104 MB; the PyTorch machine learning stack with CUDA uses over 200 MB; a Python-based data science pipeline with TensorFlow uses over 300 MB.

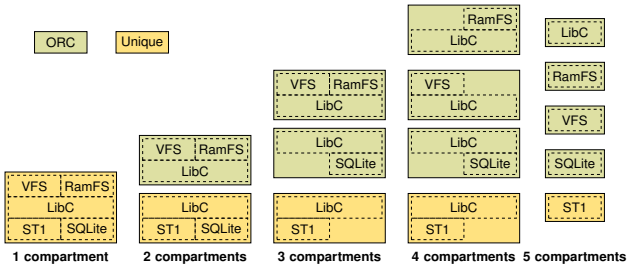


Fig. 7: Decomposition into capability compartments (SQLite)

5.4 Cost of isolation

We also investigate how increasing the number of shareable binary objects in an application affects performance. We control both the overheads introduced by the compiler pass, and those of capability-based crossings between binary objects.

To measure the cost of isolation, we execute a single compartment with the *speedtest1* benchmark and its embedded SQLite instance, and incrementally make some of its components separate binary objects (see Fig. 7). We start with one compartment that contains all components in a single binary object. Since CLS support is unnecessary here, we disable the compiler pass. We then incrementally build further compo-

nents into separate shareable objects (2 to 5 compartments), compiled with CLS support. The *VFS* and *RamFS* components correspond to internal components of the library OS; *ST1* is the benchmark binary.

Fig. 6 shows the execution times of the benchmark for different SQLite query types, varying compartment numbers. We observe that all configurations, except for 5 compartments, have only a minor performance overhead. The average difference between one compartment (everything monolithic; no CLS support) and 4 compartments (all system components but LibC are shareable objects) is 10% (median of 3%). Even when isolating all components into separate shareable objects (5 compartments), which adds many cross-compartment calls, the average slowdown is 53% (median of 39%).

Conclusions: We draw two conclusions: (1) the overhead of supporting CLS is small, especially when compared to the performance cost of cross-object calls; and (2) while the cost of cross-object calls exists, it is sufficiently small that it becomes possible to share across multiple fine-grained objects.

6 Related Work

Page-level memory sharing. 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 can also perform page sharing on disk reads. 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] uses 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 remain.

VM introspection (VMI) or graybox approaches can be used to improve the efficiency of de-duplication 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. They are treated as zero pages to improve de-duplication 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, cannot always 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 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] aggregate memory fingerprint information into a centralized control plane to determine VM placements for increased sharing and to optimize VM placement dynamically with live migration. Sindelar et al. [53] show that inter-VM sharing largely occurs hierarchically, and they propose a tree structure to manage sharing. ORC leverages 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 granularity finer 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 reason about what should be shared across tenants.

Prior work on improving scanning efficiency groups pages based on access characteristics and uses a granularity finer than pages. CMD [8] identifies page access characteristics by measuring the distribution of writes per subpage 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 approaches allow for fine-grained sharing and reduce the cost of hashing, but still lack semantic information, making sharing opportunities non-deterministic.

In modern cloud environments, it has become important to leverage large page sizes that minimize the overhead of TLB misses. The opportunity for page-level memory sharing decreases with the page size. SmartMD [27] splits large cold pages with high repetition rates for de-duplication, while re-consolidating small hot pages for improved access performance. GLUE [47] attempts to maintain large-page performance in regions that are broken into small pages for de-duplication. It extends the hardware to perform speculative large-page translation using normal-sized TLBs. While these approaches leverage finer granularities than a page, ORC has an advantage in its ability to share objects at byte granularity.

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 pages, ORC takes advantage of a semantic separation into sharable and non-sharable objects. Therefore, it is not subject to the performance overheads of existing runtime methods. Due to its use of capabilities, ORC can share objects more precise at a word granularity (i.e., spatial precision), while avoiding unintentional sharing of runtime objects (i.e., temporal precision). The loading of objects is done via a narrow interface with a small TCB, providing strong isolation.

Source code availability. The source code of ORC and our evaluated sample applications can be downloaded from https://github.com/lsds/intravisor.

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. We thank our shepherd, Malte Schwarzkopf, and the anonymous reviewers for their helpful feedback and comments.

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.