ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 46

Systems Group, Department of Computer Science, ETH Zurich

An Evaluation of Capabilities for a Multikernel

by
Mark Nevill

Supervised by
Prof. Timothy Roscoe

November 2011 – May 2012

inf | Informatik
Computer Science

Abstract

With ever-increasing numbers of cores in modern hardware and the prospect of heterogeneous architectures becoming more appealing, reliance on cache-coherent shared memory is a source for hardware design and scalability problems. To accommodate these changing requirements, multi-kernel operating systems aim to reduce the sharing in the kernel by replicating kernel state on every core.

To retain the ability to share resources between applications on different cores, we design a capability system derived from a single-core capability system used in the seL4 microkernel, but with operations extended to handle the incomplete views of the capability system present on each core. Through the introduction of an ownership property we achieve a design similar to ones found in distributed object systems. We illustrate where synchronization is still needed to ensure consistency, and demonstrate the need for an index datastructure when exchanging information about capabilities between cores.

Acknowledgements

I would like to thank Prof. Roscoe for his mentoring and input, Simon Gerber for working with me and patiently letting me discuss my ideas with him, Kornilios Kourtis, Kevin Elphinstone and Andrew Baumann for answering my questions and providing guidance at many points along the way, and many others in the ETH Zürich Systems Group for their feedback and assistance. I would also like to thank my parents for their years of support, providing a welcoming port of respite many times along the way, and my sister and other friends for their much-needed distractions.

Contents

Chapter 1

Using Capabilities for OS Resource Management

The aim of this thesis is to investigate methods of using a capability-based system for operating system resource management in a multi-kernel operating system model, and to evaluate one possible design in detail.

1.1 Motivation

The primary function of operating systems is multiplexing shared hardware resources between processes, while limiting the ability of each process to affect others undesirably, a property called "confinement". This is done at various levels by different operating systems: while some ensure only that processes cannot arbitrarily interfere, others provide many abstractions for common patterns among resource types. At its core, an operating system's kernel must therefore be able determine what resources a process may access, and must therefore track the relationships between resources and processes.

As indicated by Baumann et al. [1], mainstream operating systems generally expect memory to have a single shared address space with implicit cache coherency and all processor cores to have the same instruction architecture. Metadata is assumed to be shared, with various synchronization methods preventing race conditions. When memory is unshared or not coherent, a separate kernel must be booted on each core, and additional drivers must be used to exchange any information between these systems.

Barrelfish aims to be a scalable operating system suited for running scalable parallel applications on hardware with many cores possibly of heterogeneous architectures. It takes into account the possibility of reduced or no cache coherence and complex interconnect topologies that may be present in hardware with many processing units.

To accommodate such scenarios, we assume in Barrelfish that information is not shared initially, as is necessary with unshared and non-coherent memory. A kernel is booted on every core, and communication channels established be-

tween kernels, in turn allowing the kernels to replicate, synchronize and manage resources between one another. Where possible, applications may then build upon the kernel's communication mechanisms to establish shared resources.

With an unshared kernel on every core, kernel metadata must be replicated when the corresponding object is shared. However, the accounting information must in and of itself be stored in memory, and thus uses resources. In mainstream operating systems, this metadata memory is simply accounted to the kernel itself, and when the kernel needs additional memory, it directly allocates areas from the available physical memory. However, these allocations can directly be triggered by hardware events like page faults and other interrupt handlers making kernel memory usage unpredictable, and so the system may unexpectedly run out of memory. To remove the need for cache coherency and shared memory in the kernel we explicitly choose which regions are available to a kernel, but must therefore require that the kernel's memory usage is predictable and fully accounted for.

In the remainder of this chapter, we will see how capability systems can be used to fully account for a kernel's memory usage, looking first at various examples of such systems in section 1.2 to gain an intuitive understanding of capabilities. Because Barrelfish's capability system is heavily inspired by seL4, we provide a short overview of how capabilities are used in seL4 in section 1.3, before looking at how this design has been adapted for use in Barrelfish in section 1.4.

However, seL4 is designed for single-core systems, and so its capability system does not consider parallelism and resource sharing. In chapter 2, we therefore investigate how to extend this system in the multi-kernel model and define the semantics of capability operations in our chosen design. In chapter 3 we analyse the detailed behaviour of these new operations and their interactions. Finally, we look at how to reduce the cost of capability lookup, an operation introduced by sharing, in chapter 4.

1.2 Review of Capabilities

In abstract terms, we can describe capabilities by the following characteristics shared by capability systems:

- A form of tokens, keys or similar, which we shall refer to as capabilities, is used to reference objects in the system.
- Without any capabilities, actors do not have access to any objects.
- Capabilities can only be set from other capabilities or via particular calls into the capability system's trusted core.
- Capabilities may be dereferenced, invoked or similar. The capability system checks the validity of the specified capability and if it provides privileges to perform the action specified.

For example, many Unix-like operating systems use so-called "file descriptors" to track which processes have gained access to which files. Because these files may also be wrappers around various hardware devices, the end effect is that these file descriptors track not just access to storage on a filesystem, but

also which process has gained access to which hardware device, and what operations may be performed on said devices. In this scenario, the file descriptor is simply an index into a file descriptor table that the kernel has associated with each process. Thus, the file descriptor's value alone carries no authority, and its meaning is local to the process that has it. Sending a file descriptor to another process, e.g. by writing its raw value into a socket, has no useful effect; the other process does not gain access to the resource. Rather, the kernel must be told to copy the information in the file descriptor table into another process' file descriptor table, allowing that process to access the entry through its own file descriptor that may not match the descriptor in the original process. In fact, because the file descriptor alone carries no authority, all operations that use the file descriptor itself must be performed through the kernel.

Another variant of capabilities can be found in language runtimes implemented as application virtual machines like the JVM to ensure referential correctness. Here, memory is conceptually split into two types: data and references. References point to a chunk of metadata that precedes every data block. All data accesses by running code must be relative to a reference, with the VM enforcing that the data access is within the reference's data region by looking at the region information stored in the metadata preceding the data block. Global references and the execution stack frame reference provide entry points from which all other data is reached (a fact exploited by these systems' garbage collector for reachability analysis). To ensure references are valid, each data region's metadata contains enough information to determine which areas are references, and operations on such regions are restricted: they may only be assigned from other references, or a special "null" value, or the result of a call to the VM that creates new regions.

A solution similar to that for application virtual machines has also be applied directly in hardware: every memory word has a bit indicating if it is storing plain data or a capability. By enforcing that all memory access is based on a capability, unauthorized memory access is not possible. For example, Carter et al. [2] consider a single address-space system with 64-bit words where pointers are tagged and contain a length and permissions field in addition to their 54-bit address. All memory access must be performed through such a pointer, allowing access offsets and permissions to be checked against the pointer's information.

Capability implementations can be differentiated by how capabilities are represented to the client and where the information related to each capability is stored, both of which are influenced by the system's ability to restrict a client's access to both pieces of information. The following list presents an overview of common variants:

Tagged (with tag bits) Metadata is stored in the capability token directly, as in the system described by Carter at al [2], with a tag bit indicating which memory words are part of capability metadata. The system must be able to check every instruction for access violations. No metadata memory is necessary in the target, allowing the whole object, e.g. a memory frame, to be exposed to the client. Modifying the object in a way that affects all capabilities is however not possible, as it might require a scan of the entire memory system.

Tagged (with type system) Metadata is stored in a header preceding the capability's target object, with a part of the metadata indicating which areas of the target object are further capability tokens, e.g. using an array of tag bits. This also requires that the system can monitor every instruction for correct access, but allows more metadata to be stored then can fit in the capability directly. The capability tokens themselves simply point to the corresponding metadata block, and modifying the metadata is trivial. This system is commonly used by application virtual machines like the JVM.

Segregated Metadata is stored in protected space not accessible to clients and presented as a separate address space to each client. Capability tokens in the client are formed as addresses in the client's capability space. Special system calls must be used to perform operations on the capabilities, including copying between clients where a new copy must also be made in the receiving client's capability space. This model is used in seL4 and Barrelfish.

Password/Sparse As with a segregated system, capability information is stored in a protected space. To allow for direct copying of tokens between clients, all clients share the same capability space. This however opens up the system to capability forgery, as a client may guess capability tokens and test each one for validity, eventually gaining access to capabilities of which it never received a copy. To mitigate this, tokens are expanded in length so only a very small subset of all possible token values are valid capabilities, making it difficult to guess valid tokens. In the Walnut Kernel described by Castro et al. [3], 64-bit capability identifiers are extended with a 64-bit password that must match the password stored in the capability's metadata in protected space. Introducing a penalty for using invalid tokens further restricts a client's ability to enumerate and test token values.

1.3 Capabilities in seL4

To guarantee various properties related to security, resource usage and real-time constraints, various operating systems have attempted to formally verify their system. To make this feasible, the seL4 operating system described by Klein et al. [7] attempts to minimize the amount of code that must be verified to ensure correctness of the overall system. To separate the system into individually verifiable components, these components must be confined, limiting in their ability to affect the remaining system, and the confinement system must itself be verifiable. As described by Shapiro et al. [9], capability systems with the required properties are a sufficient mechanism to formally verify confinement.

In seL4, a capability system is therefore used to account for all system resources managed by the kernel: memory, threads, capabilities, and communication channels. After bootstrapping an initial thread, the kernel leaves management of resources up to userspace, only enforcing correctness through the capability system.

For their system, seL4 uses segregated capabilities, storing capability metadata in memory accessible only to the kernel. Arrays of capability slots that

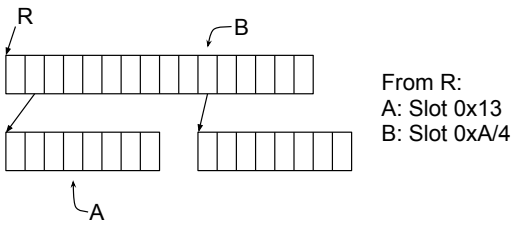

Figure 1.1: A simple CSpace example with the addresses of the two marked slots as seen from R. As shown for B, an explicit depth (number of valid bits in address) must be provided to address inner nodes.

may contain pointers to further such arrays form a directed graph of capabilities. This graph is exposed to userspace as a capability tree that can be walked by specifying an address formed as a sequence of offsets into slot arrays. When userspace provides the kernel with such a sequence in the form of an address, the kernel starts from a thread-specific root node and sequentially applies each offset to look up the next node, as shown in Figure 1.1. Thus, capabilities form an address space referred to as "CSpace" that is specific to each thread based on the root node associated with that thread.

With this system, it should now be possible to start a thread with a capability to access some region of the system's memory. We will now look at how this memory capability can be used to start further threads, and how it can be split into useful parts which may be mapped into the thread's virtual address space.

To allow the various uses of such a memory capability, seL4 introduces a retype operation. Given a generic memory capability (referred to as "untyped memory", or UM), seL4 allows this to be retyped to a mappable "virtual memory" (VM) capability, or an array of empty capability slots called a "CNode", and other types (see Figure 1.2). Additionally, the retype operation may be used to split the capability into multiple derived capabilities covering adjacent but distinct areas of the source capability. This allows a large memory capability to be split into smaller frame capabilities that may be mapped in different locations of the address space.

Retyping the same area of memory into different types can however cause problems: A "thread control block" (TCB) capability contains kernel-private information about a running thread. If this could be modified directly by userspace, threads would be able to escape the constraints enforced by the kernel, e.g. by mapping an area of physical memory to which they should not have access. However, if a region of memory containing a TCB would also be covered by a "virtual memory" capability, the thread owning the capability would be able to do precisely that.

For this reason, the retype operation enforces some constraints: Once an area of untyped memory has been retyped to a more specific capability type, that area may not be retyped again until all copies of the derived capability have been deleted.

The end effect is that memory is laid out as a tree of objects as shown in

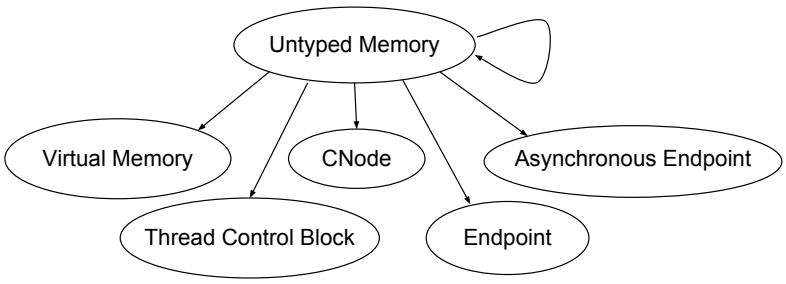

Figure 1.2: Capability types in seL4. Arrows represent permitted retype operations. The circular arrow represents the ability to retype UM into to smaller UM chunks.

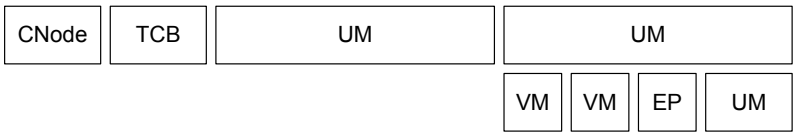

Figure 1.3: Example layout of memory in seL4. Acronyms are explained on page 7. On the right side, a UM region has been split into multiple smaller UM regions, some of which in turn have been retyped for other uses

Figure 1.3: at the top, untyped memory, possibly with more untyped memory below if the top node has been split. Below these, various other memory types forming the leaves of the tree.

To enforce the constraints described above, the retype operation needs to know if a given untyped memory capability has already been retyped. This is the case exactly when the untyped memory's node in the memory object tree has descendants. To query this information, seL4 stores the memory object tree in the Capability Derivation Tree (CDT) — called the "Mapping Database" (MDB) in Barrelfish — allowing efficient checks for descendants and copies.

In its model, seL4 defines five capability operations, defined as invocations on the CNode containing the target capability slot: "copy", "delete", "move", "mint", and "revoke". The first three operations behave as expected from their names, performing basic manipulations of capability slots. The "mint" operations is a variant of "copy" where the authority provided by the capability in the destination slot may be reduced as it is created. The "revoke" operation is more complex, allowing a capability holder to remove all other uses of the capability from the system by deleting all copies and descendants of the revocation target.

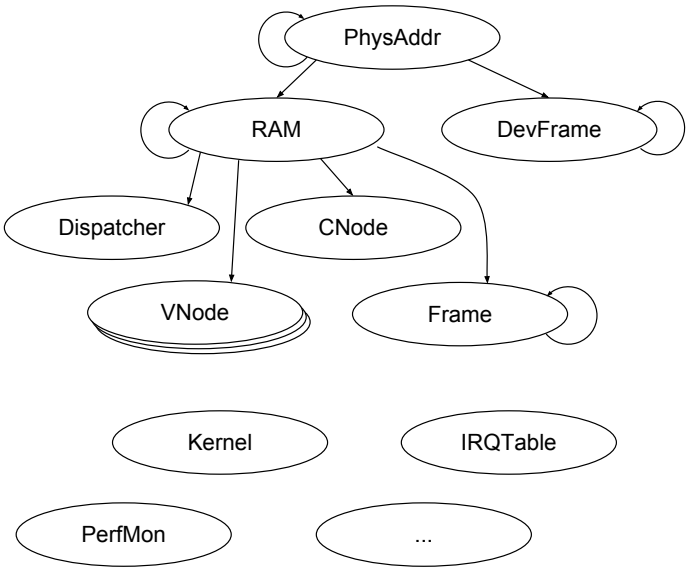

Figure 1.4: Incomplete overview of capability types defined in Barrelfish, including possible retype paths. `VNode` (page table) types for different architectures and PT levels have been stacked.

1.4 Capabilities in Barrelfish

On a single core, Barrelfish's capability system is based on the model defined by seL4. However, Barrelfish intends to handle multiple cores, multiple architectures with differing word sizes, and even heterogeneous architectures. Handling multiple cores is described in chapter 2. First, we will look at Barrelfish's capability architecture on a single core.

Types

To deal with various tasks needing privileged access to parts of hardware, Barrelfish has introduced a number of additional capability types, as shown in Figure 1.4. These capability types are specified in Barrelfish using a domain-specific language for which examples can be found in Appendix A. From this specification we can use a translation program to automatically create a number of methods and structures needed by the capability system, reducing the need for hand-written and error-prone boilerplate.

Main Memory The equivalent of seL4's "Untyped Memory" in Barrelfish is the `RAM` type. As with UM, These capabilities can be split into smaller chunks or retyped as in seL4: `Frame` capabilities can be mapped into virtual memory, `CNodes` hold capability slots, `Dispatchers` represent tasks with a virtual address space, capability space, scheduling parameters and more.

Page Tables In many operating systems, page faults are handled by the kernel or some other dedicated paging handler. Such a handler must therefore be able to allocate memory for the faulting application, and defines the policies employed by that application. In Barrelfish, applications have direct handles on their available memory via capabilities, making an external paging service unnecessary and problematic. Instead, applications are self-paging as described by Hand et al. [6]. By directly exposing hardware page table types in the form of individual capability types, the Barrelfish kernel is able to easily enforce correctness of application-built page tables. Additionally, applications are able to build page tables for architectures other than the one executing the code, allowing such an application to build page tables for other cores in an environment with heterogeneous architectures.

Device Memory Barrelfish `RAM` capabilities, the equivalent of seL4's "untyped memory", require that the memory be zeroed before it is read, to ensure that information is not leaked unintentionally. However, memory-mapped device registers should not be arbitrarily zeroed. Therefore, mappable `DevFrame` capabilities have been introduced that have no such zeroing requirement. However, these capabilities cannot be derived from `RAM` capabilities, as one would have to first create a `RAM` capability, which would zero the relevant memory. To solve this, we introduce a parent type for `RAM` and `DevFrame` capabilities, the `PhsyAddr` type. This type represents a range of physical addresses and nothing more, and is not directly useful without being retyped first.

Page tables Mapping memory in Barrelfish is done by manipulating page table capabilities. To enforce a correct page table hierarchy, Barrelfish has a capability type for each level of page table on each architecture it supports. With support for x86_64, x86_32 and ARM, this requires nine page table types to be defined.

Kernel interface As described in chapter 2, Barrelfish splits its kernel into a privileged and userspace part. The userspace performs privileged actions by invoking its `Kernel` capability.

Others Additional capability types exist for various tasks like performance monitoring and handling legacy I/O devices.

Invariants

We will next consider the invariants of this capability system. As the system is based on seL4, the invariants are similar, but must take the generalized formulation of capabilities into account.

Slots A capability slot contains exactly one valid capability, where the special "Null" capability is used to model "empty" slots.

Address ranges A capability type may be "addressable". All objects of this type must represent an address range that they cover, specified by a base address and size property in capabilities for that object. The "Null" capability type is not addressable. Note that although the primary purpose

of this property is for representing ranges of physical addresses, it may also be used for other address ranges that may exist in a system, e.g. legacy IO ranges.

Equality fields A capability type may specify any number of equality fields. All capabilities of that type must have these fields.

Copies Any number of copies of a capability may exist in the system. Two capabilities are considered copies iff they are of the same type, have the same base and size if addressable, and have matching equality fields.

Relations Capabilities may have ancestors and/or descendants iff they are of an addressable type.

Ancestors Capabilities may have at most one immediate ancestor (which may have any number of copies). The ancestor's type must be equal to or an ancestor of the capability's type. In the first case, the capability's address range must be a strict subset of the ancestor's address range, in the latter case, a non-strict subset.

Overlaps For every pair of addressable capabilities with intersecting address ranges, one capability's address range must be a (non-strict) subset of the other's.

Immutability Fields of a capability that participate in copy comparisons, i.e. type, base, size and equality fields may not be modified except when the capability is deleted in its entirety, i.e. reset to Null.

Operations

The operations also based on seL4. Because we will specify them again when considering sharing, the operations and their contracts are only loosely defined here.

Copy Create a copy of the source capability in the destination slot.

Mint Equivalent to a copy, with the possibility of adjusting type-specific capability parameters (non-equality fields).

Retype If the retype is well founded according to the capability type specifications, create descendant capabilities in the destination slots.

Delete Delete the capability in the target slot. This may require type-specific per-copy cleanup operations, e.g. unmapping pages from virtual memory. If this is the last copy, of the capability, additional type-specific cleanup operations may be necessary, like clearing slots for CNodes, page table entries for VNodes, and possibly reclaiming memory when it is no longer referenced.

Revoke Delete all copies and descendants of the capability in the target slot (leaving the target itself as-is).

Chapter 2

Sharing Resources Across Cores

With the introduction of capabilities to manage kernel resources, we are now able to account for usage of resources managed by the kernel. As in seL4, a copy operation provides the ability to copy capabilities between slots, allowing a client to grant another client access to a resource by copying the capability into a receiving slot in the target client. While this is sufficient for sharing resources, it is not safe: copying a capability is not an atomic operation, and other capability operations like revoke require a consistent view of many capabilities in the system. If these operations were run on another core, parallel reads and writes of the same capability slots might interleave and incomplete operations in the MDB might be exposed.

In Barrelfish, to limit the need for synchronization in the kernel, a separate copy of the kernel is run on every processing core, with copies not sharing any resources by default. This per-core share-nothing approach is also extended to userspace, creating a system where each core acts as a node in a distributed system.

While this solves synchronization issues in the kernel and allows simultaneously running kernels on cores of differing architectures, the system no longer supports many common scenarios in applications: sharing resource between dispatchers running on different cores is not possible. As an example, let us consider a one-to-one producer/consumer setup, where a dispatcher on one core produces data tuples, putting them in a queue, while a dispatcher consumes entries from the queue, performing further processing. To support high throughput and avoid succumbing to latency spikes on either side, a large circular buffer is usually constructed in shared memory with cache coherency ensuring a consistent view of the buffer.

Setting up shared memory requires an agreement between dispatchers on what memory is used, and mapping of said memory into the dispatcher's virtual address space. Both parts must involve the kernel: agreement needs communication, and the kernel by default enforces strict separation. To solve this, the Barrelfish kernel provides mechanisms to establish communication channels between dispatchers on different cores. As memory is managed by the kernel's

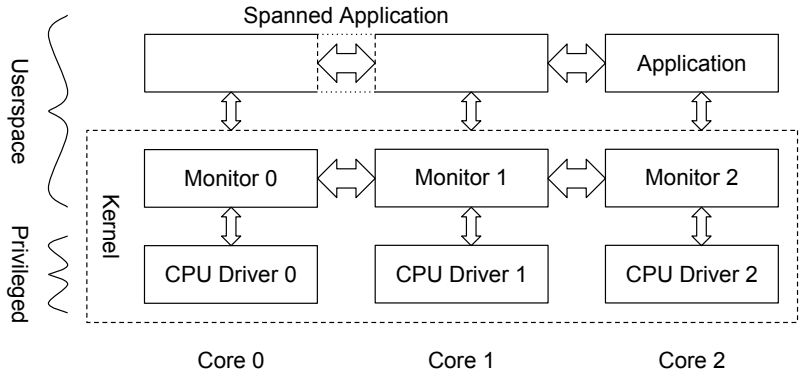

Figure 2.1: Multikernel model with split kernel

capability system, mapping the same memory in dispatchers on different cores requires the ability to share capabilities for that memory across cores.

To retain the unshared nature of kernel state, Barrelfish separates the kernel into two components, as illustrated in Figure 2.1: an uninterruptible privileged-mode "CPU driver" and a trusted user-mode "monitor", able to perform privileged calls to the CPU driver, that manages state replication and synchronization including establishing cross-core communication channels.

2.1 Ordering of Operations

With the monitor handling cross-core communication for the kernel, we are now able to pass capability information around, the basis for sharing capabilities. In a first approach, every kernel tracks which other kernels have copies of each shared capability, and ensures that operations performed on the capabilities have the correct semantics when viewed over the entire system. This last point is complicated by reordering of message delivery: although every channel in Barrelfish acts as a send-once FIFO, messages in different channels have no ordering relation, as shown in Figure 2.2.

This leads to a first problem: operations may overlap, potentially causing invariants to fail when related capabilities are affected by multiple operations. For example, two retype operations on the same source capability might, if both

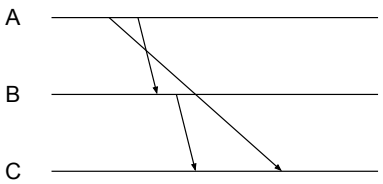

Figure 2.2: Reordering of events: single-source FIFO, not causal

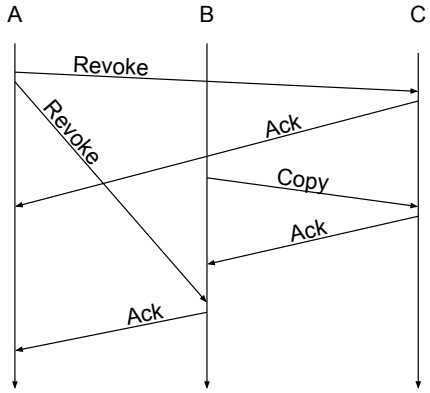

Figure 2.3: Problems with acknowledgements

executed, violate the overlap or ancestry constraints specified in section 1.4.

As described in the corresponding Barrelfish technical note [10], a simple solution using individual acknowledgements still suffers from some problems. As can be seen in Figure 2.3, a copy and revoke may be ordered such that the copy occurs between two steps of a revoke, leaving a capability in the system that should have been deleted by the revoke operation.

Another issue is not addressed by such a simple scheme: some capabilities refer to memory that is internally manipulated by the CPU drivers. Sharing these capabilities would once again require CPU drivers to synchronize among one another when user invocations require manipulations of these data structures. This conflicts with the CPU driver's goal of being a synchronization-free core to the operating system.

More generally, the design of a multi-kernel should allow for objects in one part of the system that cannot be directly accessed from another part of the system, but must instead be accessed via a suitable kernel instance. This naturally leads to the concept of object locality: every object is assigned to a core that must manage access to that object when necessary. In the capability system we model this as ownership: each capability has an owning core, which must be the same for all copies of a capability in the entire system. The owner is then used as the synchronization point when necessary for all operations on these capabilities, which we describe in the next section.

This design bears similarity to distributed object systems, e.g. CORBA or the Java RMI system presented by Wollrath et al. [11]. In Java RMI, a "stub", corresponding to a non-owned capability in our design, is used to proxy method calls across a network to server-side object. While this correspondence works well for remote invocations from non-owned capabilities, the revoke and retype capability operations have no corresponding operations in such distributed object systems. Additionally, because the objects themselves may be directly accessible from multiple nodes in the system, we have the ability to migrate the ownership of an object from one node to another without modifying the object itself.

2.2 Operations and Contracts

Using the ownership model described above, we will next consider the semantics of the capability operations in such a system.

Invariants

To understand the requirements for these operations, we will first clarify what invariants they must preserve. As we are extending the capability system specified in the introduction, we inherit the type invariants of that system. Because we have added an ownership property, we must extend the invariants to ensure correctness of that property:

Ownership Property Every capability that is not Null has an "owner" property that refers to a running kernel instance in the system.

Consistent Ownership Any two capabilities that are copies must have the same "owner" property.

Owner copies Every set of all copies of a capability must contain at least one capability where the owner matches the location. With other words, the owning core must always have a local copy.

Note that the ownership property has no bearing on its non-copy relations: a capability's relations may have other owners.

Preconditions

In the list below, we define the preconditions of Barrelfish's capability operations. Note that because of the distributed nature of the system, there is no way to ensure all preconditions are met. For example, a simultaneous revoke or delete may delete a newly created source capability before it can be used. Clients should therefore always handle the possibility of a precondition failure.

Copy The source capability is not Null and the destination core is valid.

Retype The source capability is not Null, may be retyped to the destination type, and has no descendants in the system.

Delete The target capability is not Null.

Revoke The target capability is not Null.

Postconditions

For the system to consider an operation "complete", the following conditions must hold. Note again that the invoker may be unable to verify this as further operations in the system may break the postconditions first.

Copy The destination core has a copy of the capability and the invariants of the system hold.

Retype The specified descendants exist in the target slots and the invariants of the system hold.

Delete The source capability is Null and the invariants of the system hold.

Revoke All copies and descendants of the source capability have been deleted and the invariants of the system hold.

2.3 Behaviour

We will describe the behaviour of the capability operations in such a system as seen from a global system view in pseudocode below. The operations are as listed in section 1.4, with one difference: we omit implementing a distributed "Mint" operation as it can be achieved with a distributed copy followed by a local mint operation.

First, we will clarify the semantics of the pseudocode. The operations use slots: the storage location for a single capability. An empty slot is equivalent to a slot containing a `Null` capability. Every capability — and thus every non-`Null` slot — has an immutable location and an owner, as described above. An individual capability is considered "local" if owner and location are the same, and "foreign" otherwise. When assigning to a slot *dest* with "←", we copy the capability metadata into the destination slot and update the MDB on *location*(*dest*) and any other tracking information (e.g. memory mappings) accordingly.

Copy

The copy operation must simply create a new copy in the target location, making sure that the new copy's owner is set correctly.

Algorithm 1 copy

```
operation COPY(cap: slot, dest: slot)
    if dest is not Null then
        fail
    end if
    dest ← cap
    if owner(cap) is location(dest) then
        set dest to "local".
    else
        set dest to "foreign".
    end if
end operation
```

Retype

To retype a capability, we must check that no other capabilities in the system conflict with the retype. If no conflict is found, the retyped capability is created in the destination slot. As the owning core must always have a copy and we do not want to create capabilities not explicitly requested, the target core must also become the owner of the new capabilities.

Algorithm 2 retype

```
operation RETYPE(cap: slot, region: range, type: captype, dest[1]: slot)
    if dest is not Null ∨ retype(cap, region, type) is not valid then
        fail the RETYPE operation.
    end if
    if any descendants exist locally or remotely then
        fail the RETYPE operation.
    end if
    dest ← local retype(cap, region, type) on location(dest).
    set dest to "local".
end operation
```

Delete

Because the owning core must always have a copy of the capability, the delete operation gets complicated when applied to the last copy on the owning core. In this case, if other copies of the capability still exist in the system, ownership must be transferred. This is further complicated because not all capability types support changing ownership: capabilities of some types, e.g. CNode and Dispatcher capabilities, represent CPU driver state, and would require more synchronization to migrate from one CPU driver to another.

Algorithm 3 delete

```
operation DELETE(cap: local slot)
    if last copy on owner(cap) then
        if cap is not moveable then
            for all foreign copies on all cores do
                DELETE (copy).
            end for
            do cleanup (last copy deleted).
        else
            dst ← find a foreign copy of cap.
            if dst exists then
                CHOWN (dst).
            else
                do cleanup (last copy deleted).
            end if
        end if
    end if
    cap ← NULL
end operation
operation DELETE(cap: foreign slot)
    cap ← NULL
end operation
```

[1] In both Barrelfish and seL4 it is currently only possible to retype a capability's entire region, optionally splitting it into multiple parts, creating multiple output capabilities. For the sake of simplicity, we only create one output capability per retype, allowing retype operations to specify a single sub-region of the source capability that is used for the output capability. The split operation is equivalent to performing multiple subregion retypes in the same transaction.

DELETE makes use of an internal CHOWN operation. This operation simultaneously updates the owner for all copies of the given capability such that the given capability becomes "local".

Algorithm 4 chown

```
operation CHOWN(cap: slot)
    set owner(cap) to core(cap) for cap and all copies of cap.
end operation
```

Revoke

We define revoke recursively: for each descendant, revoke and delete that descendant. Simultaneously, delete all copies of the target capability. This is equivalent to the single-core definition of revoke; the complications arise from the distributed nature of deleting the last copy of descendant capabilities. This is discussed in section 3.3.

Algorithm 5 revoke

```
operation REVOKE(cap: local slot)
    for all immediate descendants on all cores do
        REVOKE descendant.
        DELETE descendant.
    end for
    for all copies on all cores do
        DELETE copy
    end for
end operation
operation REVOKE(cap: foreign slot)
    CHOWN (cap).
    REVOKE (cap).
end operation
```

Invoke

Here, we have a significant departure from the single-core model. For invoke, we expect all invocations to behave as if going through the owning core, at least conceptually. In practice, not all invocations on foreign capabilities will need to go through the owner: a "frame identify" invocation simply returns information about a frame capability to the caller. Since this information is present in foreign capabilities, the invocation can be processed locally without communicating with the capability owner.

Algorithm 6 invoke

```
operation INVOKE(cap: slot)
    perform invocation on owner(cap).
end operation
```

2.4 Transactions

To reduce complexity of client code, we also specify a limited form of transactional semantics: An operation must succeed in its entirety or have no effect. This does not imply that intermediate steps are not visible: for example, once a revoke has been successfully initiated, that revoke must continue to completion, but we permit a view of the partially completed operation as long as all invariants remain intact in such a view.

Because of the parallelism in the system, a client cannot test if an operation's postcondition is fulfilled: by the time the test is executed, another operation may have changed the system so that the test would fail. Instead, we simply specify that an operation is complete exactly when its precondition is fulfilled, and require that operations always progress towards that state so that they must eventually complete.

Chapter 3

Detailed Analysis of Behaviour

Forcing all operations on a capability and its copies to go through a single core does not however solve all problems of synchronization: the operations "revoke", "retype" and "delete" all require a consistent view of the capability's relations. We would therefore like to analyse further how capability operations interact.

3.1 Interference Between Operations

First, we will consider each pairwise combination of operations to determine how invariants and postconditions may be violated and what must be done to fix this while also preserving the transactionality guarantees.

Retype

The retype operation must check a capability for descendants in the entire system. Copy operations do not affect this: copies of the source capability do not affect the operation, while copies of the descendants can only increase the number of descendants, leaving the fact that there are descendants intact. Thus, copy and retype do not interact.

Other retype operations also cannot cause problems: retypes on the source must go through the same owner core, where they are synchronized. Retypes on descendants can only create new indirect descendants in areas already covered by existing descendants, leaving the configuration of immediate descendants unchanged.

Delete, however, presents a problem: deleting the last copy of a descendant may affect the outcome of the retype check. We resolve this by marking the descendant in question as in-delete, and abort the retype when it encounters such a marked descendant. While this may force revokes that have no effect, it will never allow an illegal retype.

Finally, retype is affected by revokes; we will discuss this when looking at revoke.

Delete

The delete operation must dispose of any per-copy state, and perform a final object cleanup when the last system-wide copy is deleted. When deleting the last owned copy, the delete operation must also check for other copies in the system and migrate ownership accordingly, so that the capability does not become ownerless.

Creating new copies does not affect per-copy cleanup, while changing ownership is directed by the owning core and can therefore be synchronized.

As stated, deleting the last copy in the system requires an additional cleanup step that must be performed exactly once. Creating a new copy during the final delete would interfere with this as the cleanup step might be executed a second time when deleting the newly created copy. However, in this case both operations are acting on copies of the same capability, and must therefore be handled by the owning core, where they can be synchronized.

For the same reason, deletes on the same capability are already synchronized. However, deletes may involve other capabilities during the final cleanup: deleting CNodes may cause a cascade of deletes, while deleting any RAM-derived type causes an unreferenced memory check for reclamation.

In the CNode capability case, because it is the last capability copy, any attempt to access the CNode's slots in CSpace must go through the CNode itself, and can thus be blocked. However, the capabilities in the CNode's slots may also be found via the MDB, meaning they must be marked as being deleted at the same time the original deletion target is marked. A second issue when clearing slots for an in-progress CNode delete is that graph of CNode capabilities may contain cycles. If this cycle contains the capability that is the deletion target, recursively clearing CNodes in the cycle would deadlock, as that capability is used as the starting point for the delete and must therefore not itself be deleted until the rest of the operation has completed. The mark-and-sweep algorithm used to resolve these issues is described in section 3.3.

Handling memory reclamation is discussed in section 3.2.

Revoke

The revoke operation may affect many capabilities; not only all descendants and copies of the revocation target, but the capabilities contained in any CNodes among those descendants, possibly causing cascaded deletes.

To understand where revoke may interfere with other operations, we will look at an example of a revoke and retype across three cores as shown in Figure 3.1.

Three capabilities exist in the system: v at the top on core 1, t on core 2 is v's descendant, and a third capability x on core 3 that is a descendant of t. Two operations are launched, a revoke of v on core 1, and a retype of t on core 2. The revoke sends a notification to cores 2 and 3, while the retype operation only needs to notify core 3. Core 3 receives the revoke first, performing the corresponding delete. Next, the retype operation arrives on core 3, and is acknowledged as valid. Core 2 receives the acknowledgement and performs the retype, creating a new capability n, before receiving the revoke notification from core 1 and deleting both t and n.

In this scenario, both operations succeeded (for each operation a point was

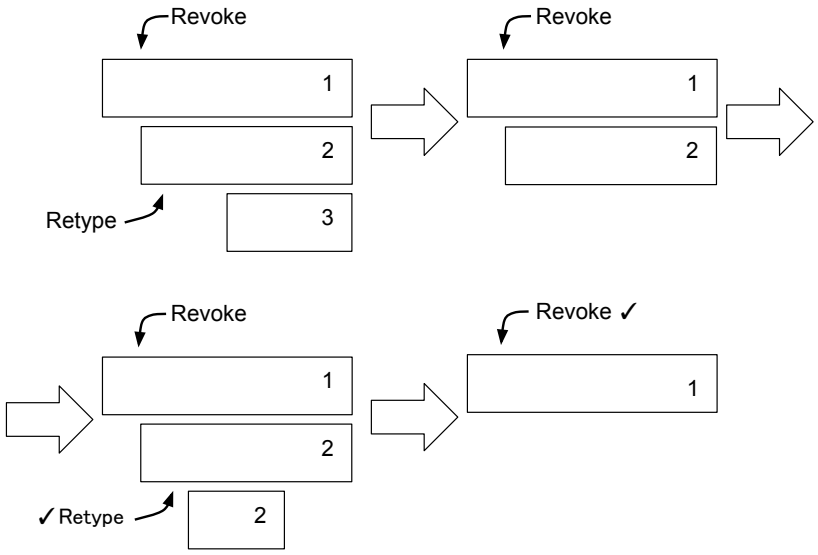

Figure 3.1: Parallel revoke and retype. Numbers indicate cores where capability is located. Core 3 handles the revoke while the retype on core 2 is still outstanding, but the retype results are deleted by the revoke once created.

reached where the postcondition was fulfilled and it could thus terminate) and no capability invariants were broken. However, the retype operation received a view of the system caused by a partially run revoke. We permit this situation: no invariants of the system are broken, and the client performing the retype must be able to handle unexpected deletion of the resulting capabilities just as easily as unexpected deletion of the retype's source capability.

As with deletion, the target of a revocation may be among the capabilities indirectly deleted, whether because it is in a slot of a CNode being revoked, or in a CNode affected by cascading deletes. This is discussed together with delete in section 3.3.

3.2 Memory Reclamation

When deleting the last copy of a capability in the system, it is possible that no application is left referencing a region of memory. If no further action is taken, this memory will leak: no component in the system is informed of its availability, and it will therefore never be reused.

This situation could be averted by requiring dispatchers to send unwanted capabilities back to a capability pool, whether in the monitor or implemented as an independent service. However, determining if a capability is no longer needed may be complicated, especially if that capability is shared between dispatchers. It also does not solve the fundamental issue of deleting the last capability for some memory, which may still occur by accident or due to a misbehaving user.

Alternatively, if we introduce such a memory pool, the pool could simply keep a copy of all capabilities it distributes, so that it could reuse these capabilities when no other references remain. Nonetheless, the reuse of otherwise unused memory regions is not trivial: the memory pool must either continuously attempt to retype the regions memory it manages to determine if they have become available, or the system must notify the pool explicitly when necessary. Depending on the pool's implementation it might also be possible for a misbehaving dispatcher to revoke any capability it receives from the memory pool, defeating the pool's ability to monitor that memory for availability.

In both solutions, dispatchers still have a mechanism to delete the last capability for regions of memory. So instead of trying to avoid this, we can instead attempt to detect unreferenced memory in the capability system and notify the memory management system, allowing it to return that region to the pool of available memory.

This design is similar to garbage collection in many language runtimes, and implementations can be either lazy (as is common with language runtimes) or eager.

Scanning Garbage Collection

Detecting unreferenced memory is not necessarily time-critical: as long as the memory is reclaimed eventually, systems with only occasional allocations should not suffer. Thus we can use a lazy form of garbage collection, scanning memory either periodically or on demand. When the memory pool is empty and receives a memory request, the pool can also trigger an immediate scan.

Analogous to language runtimes, this makes deleting capabilities simple, as no action is required, but may add a continuous overhead to the system, and may slow down allocations significantly when a collect is required.

Immediate Collection

If we would like to avoid the background overhead of periodic scans, we can try to detect unreferenced memory as soon as possible, i.e. when a capability is deleted. This is not usually possible in language runtimes as it would still require a scan of all references in memory. In our system, we can instead scan the MDB in every kernel for relations.

Scanning for all relations in the entire system on every delete would create a large overhead, but is not necessary. Instead, we can hook into the delete operation as defined in chapter 2: Where the operations performs cleanup actions for the last copy in the system, we begin our check for remaining relations in the system.

In a first approach, we check for remaining ancestors and descendants; first locally where the final delete occurs, then in the entire system. If neither ancestors nor descendants are found, the memory is reclaimed.

As shown in Figure 3.2, this is a conservative approach that may still leak memory. A workaround of searching forwards and backwards for more unreferenced memory as shown in will ensure that all memory is eventually reclaimed, but may still leave regions of memory unusable for extended periods. This may in turn be resolved by scanning the entire deleted region for areas without descendants, introducing even more overhead to delete operations.

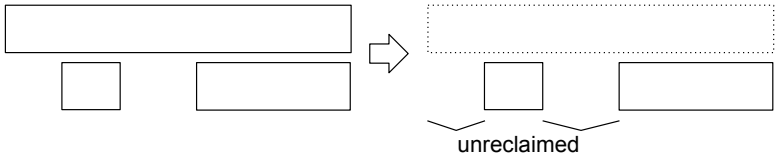

Figure 3.2: Naïve reclamation leaks when a only parts of a deleted region have descendants.

Combined

Finally, the possibility exists of combining scanning and immediate methods: a scanning garbage collector that is only triggered when simpler methods are unable to determine if a delete requires memory to be reclaimed.

Fragmentation

When memory is reclaimed as described above, a capability is produced for every unreferenced region. As our capability system is only able to split regions into smaller regions using the retype operation, memory may gradually fragment until it is no longer possible to allocate large chunks of contiguous memory.

Lazy collection might partially alleviate this problem as more fragments may become unused while the collector waits and which the collector may recreate as a single capability. This would however require tuning of the collector's scan frequency to application behaviour and could still be circumvented.

Instead, we can modify the system to allow merging of capabilities. Such a merge operation would require some restrictions:

- The merged capability must not cover parts of memory that the input capabilities did not cover. Therefore, the input capabilities must be contiguous and the merged capability must start precisely at the smallest base address and end at the highest end address of all input capabilities.
- The merged capability must be a legal ancestor of all its inputs. It must therefore have a type that is a common ancestor the inputs.
- Merging capabilities must not modify the objects corresponding to the input capabilities.
- Creating a capability of a more general type may allow operations not previously available to the caller on the more specific input types. Therefore, the only legal merges are when the input and output types are the same and a self-retype loop exists for that type, as is the case for the `PhysAddr`, `RAM`, and `Frame` types.
- Similarly, creating capabilities of the same type but larger size allow operations not previously available to the user: if the new capabilities already has copies in the system, the user may revoke this new capability, thereby deleting those copies. Merging capabilities must therefore not be possible when would-be copies of the resulting capability exist.

Algorithm 7 merge

```
operation MERGE(left, right, dest)
    if dest is not Null or merge(left, right) is not valid then
        fail the MERGE operation.
    end if
    dest ← merged(left, right)
    if dest has any copies locally or remotely then
        set dest's locality according to those copies.
    else
        set dest to "local".
    end if
end operation
```

3.3 Delete Cascades and Reachability

The possibility of shared capabilities adds significant complexity to deletes and revokes, as we will see in this section. Let us first consider delete on its own.

When a capability is to be deleted, three cases present themselves: In the simplest case, the capability has local copies or is foreign. In this case, the ownership of the capability is not lost upon delete, and so the capability can be reset by the CPU driver directly without need for any cross-core negotiation.

In the second complex case we are deleting the last copy of a capability with local ownership, but with remote copies. If possible, ownership must be transferred to another core that has a copy of the capability using the "move" operation. If this succeeds, the capability is now foreign, and can be deleted safely. On the other hand, if ownership cannot be transferred for this capability type, all copies of the capability in the entire system must be deleted, and the initial delete continues in the next case.

The final case is when the capability is the last copy in the entire system. In this case, any clean-up actions for the object represented by the capability must be performed. For a RAM-derived capability, this may mean that the kernel reclaims the unreferenced memory and sends it back to the memory server. In the case of a dispatcher, that dispatcher is terminated. And in the case of a CNode, all the slots of the CNode must be cleared. This last case is where complexity arises: If the initial CNode contains another CNode capability that also has no copies, the same slot clearing must be performed on that CNode prior to deletion. This can therefore result in a cascade of deletions, a complex and long-running operation which at any point may re-enter this complex third deletion case. Additionally, the chain of to-be-deleted CNodes can circle around, with the CNode containing the original capability also scheduled for deletion.

Before we look to solve this, we will also take a look at how this affects deletions that occur during revocations, whether due to the revocation or due to a separate delete request.

Revocation of a capability deletes all copies and descendants of that capability in the entire system. This implies that the capability itself must remain referenceable during the entire revoke operation, which in turn implies that the CNode containing the capability must not be deleted until the entire operation can be executed without needing to reference the original capability.

This complexity is not entirely caused by sharing capabilities, but the need at any point to interrupt the operation and run a cross-core agreement protocol makes it impossible to store temporary global state in the CPU driver while the operation is running; any state must be stored in the capabilities being deleted or revoked.

Solution

Our solution is to clear the capability graph of capabilities for objects that do not contain capability slots or that can be trivially deleted. Once this is complete, we have a self-contained graph where all nodes must be deleted. We can therefore explore this graph, adding all nodes we find to a deletion queue, which can then be deleted in a single loop.

1. (revoke only) Find all descendants. For every descendant, perform the "delete" operation.
2. To delete a capability:
    a) When deleting the last copy of a Dispatcher capability, clean up the dispatcher, leaving the two capabilities stored in the `dcb` struct intact.
    b) When deleting either a CNode capability or the last copy of a Dispatcher capability, mark the capability as deleted without clearing it, and insert it at the end of a singly-linked "clear" list stored within the extended region of the capability slot.
3. Work through the "clear" list, performing a "delete" as described above on every slot contained in the objects referenced by the list entry. Then place the entry in a "delete" list.
4. Walk through the delete list, performing the final clean-up of every entry in the list.

The corresponding capability state machine is shown in Figure 3.3. Copy and retype are not included; both operations simply put the capability in the locked state until the operation completes or fails.

In the algorithm described, both revoke and delete may require locking or marking many capabilities. Meanwhile, other operations may also be trying to lock some of the same capabilities. To avoid deadlocks between multiple revokes and/or deletes, we simply merge the operations, and consider all deletes and revokes locally complete when there are no remaining marked capabilities. For copies and retypes that only lock a single capability and its copies, we simply wait until the lock has been released before locking it again for deletion.

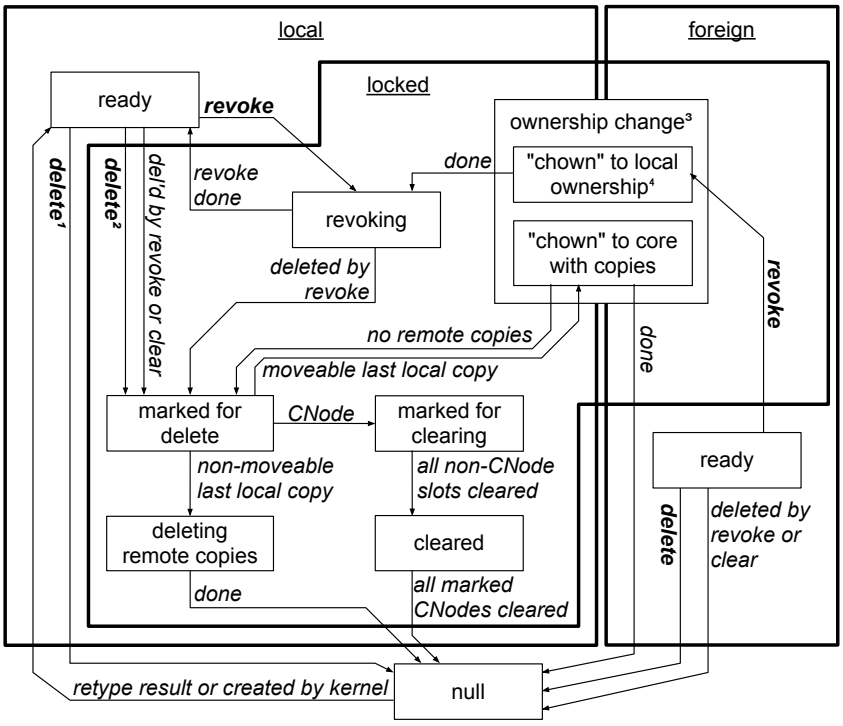

Figure 3.3: Per-capability slot state machine for deletes and revokes. Actions in bold are user-initiated. [1,2]When local copies of a capability exist, delete can directly null the capability slot. [3]When changing ownership, the invariant of having exactly one owning core that is equal for all copies in the system may be temporarily violated. [4]This may fail, returning the capability slot to the ready state.

Chapter 4

Capability Lookup

To perform distributed capability operations across multiple cores, we first need to make sure two monitors can exchange information about capabilities, and in particular, exchange the data stored in each capability. For this we must serialize capabilities on the sender, and deserialize them again on the receiver. Serialization between differing architectures on heterogeneous systems requires that any transformation of physical address space layouts be applied to exchanged capability metadata, but this is not explored in this thesis. Instead, we assume that copying the bytes corresponding to a copy of a capability is sufficient for serialization, and therefore that all nodes have the same view of physical memory. However, the receiver must take an extra step: As noted in section 1.3, the kernel maintains an internal database, the mapping database (MDB), containing the ancestor, copy and descendant relationships between all known capabilities. For the received capability to be used and managed properly, it must first be inserted into this database.

With the introduction of cross-core capability sharing, we have also introduced the need for new query types to be performed on this database:

- Upon receiving a serialized capability with the copy operation, that capability must be inserted into the MDB. Because the MDB itself is used to lookup up relations, this insertion must be done without any prior knowledge of existing ancestors, copies or descendants.
- Checking if a capability retype is legal requires us to check if a capability has descendants. Again, the capability to check is received in serialized form without knowledge of relations. One solution would be to insert the capability in the database and then check for descendants, deleting the capability again when the check is complete. However, this creates a temporary capability in the system not specified in our high-level operations. While this might not have an effect on the system, we try and avoid doing it; we should therefore support querying if a capability has descendants without inserting the capability if possible.

- Deleting the last owned copy of a capability requires that the system find another copy and mark it owned. Again, performing this check by inserting the capability creates a new copy not accounted for in the design, especially problematic in this case since we are trying to check for the existence of copies. Thus, it must be possible to check for copies without inserting the capability.

- Memory reclamation as described in section 3.2 needs the ability to find regions of memory that are not covered by any capabilities in the system.

  When implemented eagerly on deletion of a capability, this requires that the entire system be checked for ancestors, copies and descendants. Copies are tracked by design, and check for descendants has already been covered, but checking for ancestors is a new requirement. This check must be performed without creating any visible copies of the deleted capability.

  We may alternatively choose to scan memory for unreferenced regions. This can be done by stepping through the regions covered by capabilities, in which case we must at least be able to search the system for a starting capability, and forward or backward siblings of a given capability.

- Revoke deletes all of a capability's descendants and copies. On the core where the revoke is executed, copies and descendants can be found by searching relative to the revocation target. On other cores, there may not be a copy of the revocation target, so we must instead query the source capabilities address range for descendants.

- Remote invocations need to find a local copy of the invoked capability from its serialized form.

In seL4, the MDB is stored as a doubly-linked list, representing the preorder-DFS through the hierarchy of capabilities. This data structure allows easy insertion of a capability given its immediate ancestor or a copy, and easy checking for existence of copies and descendants.

But when no relations are known beforehand, finding the position to place a new capability requires a $O(n)$ linear scan through the list, as does finding ancestors and descendants of a capability given just the capability's value. This operation is performed in the non-preemptable kernel, creating a scheduling hole that is problematic for real-time applications.

To reduce the complexity of operations described above, we replace the MDB's linked list with a more suitable search data structure.

4.1 Review of Search Datastructures

We shall first consider hash tables. Hash tables provide $O(1)$ lookup with high probability, and $O(1)$ insertion (amortized or with high probability). However, we face multiple problems in employing a hash table for the index: Firstly, hash tables cannot represent the hierarchical relationships between capabilities. One would still need to maintain additional metadata to find immediate ancestors and descendants. However, it is not enough to simply add a pointer linking to each, as individual copies of the relations may disappear; one would have to check and update these pointers on every delete. Another issue also hinders

the adoption of hash tables: While a hash table's space complexity is $O(n)$, there is no direct relationship between memory used by the hash table and elements in the hash table. One would therefore need to allocate additional memory outside of the capabilities and CNodes. However, the MDB is stored and maintained by the CPU driver, and the whole purpose of the capability model is precisely to avoid such allocations in that part of the kernel.

We next consider designing a custom data structure with direct links for all the relationships to make queries $O(1)$ where possible. Thus we are looking for a tree structure that maps directly to the capability hierarchy, with direct links in each node to ancestors, copies, descendants. Additionally, because a node can only link to one immediate descendant, all immediate descendants need to be connected in a "sibling" list.

To look up a capability, we recursively walk down the hierarchy: starting at the first root, we walk the sibling list to find a root node that covers the target capability. If the found node does not match the target, we recurse: starting at the first immediate descendant, we again walk through the list of siblings, and so on. This algorithm presents a first problem: once again, we have a worst-case of $O(n)$. To solve this, we can replace the sibling list with a sibling tree with an ordering based on each capability's base address.

We now have fast lookup, but at the price of having a complex algorithm. For example, deleting a node may require the tree of the node's immediate descendants be merged into the node's own sibling tree. Additionally, we still have a problem that we had in the hash table-based solution: a capability may have many copies, any of which may be deleted; pointers to relations must be updated when the specific copy that is their target is deleted.

The fundamental cause for the pointer maintenance problem comes from the reduction of a many-to-many relationship between all copies of a capability and all copies of it's immediate ancestor to a direct many-to-one relationship. This reduction is necessary when using direct references to relations because many-to-many relationship must be stored externally to both sides of the relationship, but the CPU driver is not able to allocate space external to capabilities.

To circumvent this problem, we simply must not directly store the relationships at all. Instead, we create a searchable index that is able to efficiently answer the required queries. However, the space restrictions remain: the index must be stored within the capabilities themselves. We thus look to a class of data structures that have a direct correspondence between nodes and elements: search trees. With a search tree, we can look up capabilities by value, and find copies quickly by placing them sequentially in the tree's ordering. Not all queries are as simple, however: if we place a capability's first descendant close in the ordering (as in seL4's preorder-DFS), the ancestor will be further away in the other direction, and reverse. To compensate, we convert the tree to an interval tree using the augmentation technique described in Cormen et al. [4, p. 311 – 317], which allows us to search for capabilities covering a address range, which we employ for the ancestor query.

A common choice of search tree for databases and filesystems is the B-Tree. B-Trees are balanced, can be very shallow compared to other tree types, and are useful when the node size can be tuned to some block size of the underlying storage system for improved performance [8]. Since there is at least one element for every B-Tree node, there will always be a capability available to store the node, fulfilling our space requirement. However, knowing where to store the

node is not so simple; elements can be pushed up and down in a B-Tree, or even be removed without changing the number of nodes in the tree. Thus, a B-Tree implementation would have to be able to migrate tree nodes from one capability slot to another as slots containing tree nodes become unavailable. Additionally, B-Tree nodes are fairly large: the nodes of a 2-3 B-Tree, the smallest viable B-Tree degree, contain 6 pointers, for a total of 48 bytes for 64-bit architectures, and 24 for 32-bit architectures. This area must be present in every capability, used or unused, so we would like to minimize its size.

Because of the complexity of migrating B-Tree nodes between available capabilities, we will also look at binary trees variants where each element is a node. Because of this correspondence, a node is removed exactly when its element is removed and vice versa, meaning no node migration is necessary. Additionally, the tree needs only a small amount of data per node: two child pointers, a parent pointer and usually a small amount of metadata, e.g. the depth, sub-tree height or "colour" of the node (for red-black trees), totalling 25 and 13 bytes for 64-bit and 32-bit architectures respectively.

For the sake of simplicity, we have chosen to implement the index using an AA tree. This tree, an isomorphism of a 2-3 B-Tree, guarantees that the deepest leaf is at no more than twice the depth of the shallowest leaf, and that that deepest leaf is the rightmost leaf in the tree, the last element in the ordering. Additionally, we have chosen to drop the parent pointer from tree nodes so we can use the space for other purposes, reducing the node size to 17/9 bytes (64-/32-bit architectures). This has the effect that retrieving the predecessor or successor of a node is no longer $O(1)$ on average, but may instead require a search from the root for the next element in the ordering. The performance effects of this modification are analysed in section 4.4.

4.2 Ordering

To implement a tree-based index, we now need to define an ordering. This order must be defined such that the operations defined earlier can be performed efficiently. To find all copies of a given capability, we would like an ordering where copies are immediately adjacent to one another. Similarly, to move up and down in the hierarchy, relations should also be in close proximity. In essence we would therefore like an ordering similar to the previously used preorder-DFS, except that any two capabilities must be comparable. From this we can obtain these first constraints on the ordering:

- Memory capabilities for an area with a higher base address must come after capabilities for areas with a lower base address.
- For memory capabilities starting at the same base address, the smaller capability must come after the larger capability.

From these, we can determine an initial requirement: both base address and size must appear in the ordering, and the base address must have a higher priority. Also, as smaller sizes must appear later, sizes must be in descending order. Thus, we have this initial tuple for lexicographical ordering:

$$(base, -size)$$

Next, we look at the relations between types. When two memory capabilities cover the same area, but the second is derived from the first, how to we place these capabilities in the ordering? Since the second is a descendant, it should appear after the first. However, any smaller capabilities covering a subregion of these capabilities must be descendants of both, and must therefore appear after both. Thus, we need to have an ordering by the type hierarchy that appears between *base* and $-size$ in the ordering tuple.

How do we create such a type ordering? For this, we must first constrain the capability type hierarchy to a tree. This allows as to define a partial ordering between types, which we can use in our global ordering. This opens the question how to use the partial ordering when comparing capabilities for which the partial ordering is not defined. Here, we are saved by the nature of this hierarchy: Retyping memory capabilities can only happen "down" the hierarchy, and thus all capabilities with the same base address must lie on a single path from the hierarchy root to a leaf. Since we have already concluded that we must apply the type ordering *after* the base address ordering, we will only ever be comparing types for which the partial ordering of types is defined. Thus we arrive at this ordering tuple:

$$(base, type, -size)$$

One important aspect of capability types remains to be considered, and was briefly mentioned in the previous paragraph: the type hierarchy is not a tree, but a forest, containing types that do not cover any area of memory. However, all such types lie in trees separate from the tree of memory capability types, which leads to a simple solution: We order the type trees themselves, and use this ordering to resolve comparisons between unrelated types. This leads to the following ordering, with base and size set to a single value (zero for our purposes) for non-memory capability types:

$$(tree, base, type, -size)$$

Additionally, all capability types can have fields designated to be used for equality comparisons between capabilities of that type. Since we have already handled comparing different types, we can just add these fields to the end of the ordering tuple:

$$(tree, base, type, -size, eq \ldots)$$

We face one final issue: each copy of a capability would be considered equal with this ordering. But for insertion into the index, copies must also have a stable ordering amongst each other. For this, we add a tie breaker, using the capabilities' in-kernel address:

$$(tree, base, type, -size, eq \ldots, address)$$

We now have an index into which we can insert and delete capabilities. Let us now analyse how to perform the operations we require.

First, various operations need to find all copies of a capability, or check if copies exist. By definition, a copy differs only in its address, which is the last element in the ordering. Thus, all copies will be siblings in the ordering,

so we can find all copies by iterating forwards and backwards from the initial capability until we reach the edge of the tree or find a non-copy. All capabilities returned in this fashion will be copies.

Next, revoke and retype need to check if a capability has any descendants. By construction of the ordering, if a capability has any descendants, the first will be located immediately after all copies of the capability. We can therefore search forward past all the copies, and return true if the next capability is a descendant.

4.3 Range Queries

By augmenting the tree with an end interval as described in Cormen at al. [4, p. 311 – 317] we gain the ability to perform searches for ranges. Note that looking up capabilities for a single address is also a range search, as an address may be "covered" my multiple regions in the ordering, e.g. when a capability for memory containing the address is preceded by siblings not covering the address before which there is an ancestor that again covers the address.

More concretely, we use range queries in two scenarios:

- When looking for a capability's ancestor, and
- when looking up capabilities during a frame unmap based on the physical address stored in page tables as described by S. Gerber [5].

To search for a target capability's immediate ancestor, we can first search for a capability earlier in the ordering, and check if it is an ancestor, in which case it is the immediate ancestor. If this is not the case, we encountered one of two situations: The target capability has no ancestor, or the target capability has an ancestor but that ancestor has descendants that precede the target capability. Using a range query, we can search for the smallest capability that covers the starting address of the target capability, i.e. that contains the range from target.base−1 to target.base+1.

4.4 Performance Evaluation

Setup

We will now evaluate the performance of our implementation. Where possible, we will use an MDB implementation following seL4's doubly-linked list design as a comparison to our augmented AA Tree. Both implementations are evaluated by running microbenchmarks in Barrelfish, running in userspace, on the machine described in Table 4.1.

To gain an overview of MDB performance, we first run randomized benchmarks. Each measurement is run 1000 times. In each run, the MDB is reset and a capability slot array of the indicated size is filled with naturally-aligned RAM capabilities. Measurements taken in Barrelfish indicate that a booted system may have roughly 3000 capabilities, of which 99% are `PhysAddr`-derived capabilities, 20% have copies, 10% have ancestors and 5% have descendants.

For our benchmarks, capabilities are generated randomly such that approximately 10% of capabilities are copies of a capability in the other 90%

Processor	2x dual-core AMD Santa Rosa (Opteron 2200)
CPU speed	2800 MHz
L1 cache	64KB icache and 64KB dcache per core
L2 cache	1MB per core
TLB size	1024 4K entries
Main Memory	8GB

Table 4.1: Evaluation hardware

(i.e. roughly 20% of all capabilities have copies). To approximate a "regular" distribution of capabilities, we generate the remaining such that many small capabilities but only few large capabilities are created. The remaining we therefore generate such that the probability of creating capabilities of a given power-of-two size is proportional to the negated power, i.e.

$$\mathrm{P}\left[\log_2(size) = x\right] = \begin{cases} x < max & 2^{-x-1} \\ x \geq max & 0 \end{cases},$$

where max represents the total amount of memory.

Results

In figure Figure 4.1, we can see the expected $O(n)$ insertion time characteristic of the linked list implementation, along with the $O(\log n)$ behaviour of the search tree. In the best case, however, the linked list fairs much better: once the insertion point is found, insertion is simply an update of four pointers and therefore $O(1)$. Therefore, if the insertion point is near the beginning of the list, insertion is faster than in the tree, where finding the insertion point is always $O(\log n)$.

With removal from the tree shown in Figure 4.2, the linked list is $O(1)$, needing just four pointer updates (as with insertion when the insertion position is known). Meanwhile, the tree implementation is $O(\log n)$: not only may rebalancing be necessary during removal, but the lack of "parent" pointers requires a regular search for the removal target to build a stack of parents that need updating.

Iteration as shown in Figure 4.3 also has the linked list performing better, as an iteration step is a single pointer dereference. At 1024 capability slots à 64 bytes, the L1 cache is filled, and a performance drop occurs when proceeding to 2048 slots. Meanwhile, the search tree shows two modes, with almost an order of magnitude between them. This demonstrates the cost of leaving the parent pointer out of the tree nodes: when the successor can be found by traversing down the tree, the cost is that of a small number of pointer dereferences. When the successor is reachable this way, a regular "search" for the next element in the ordering must be performed.

The check for descendants shown in Figure 4.4 shows consistent $O(\log(n))$ behaviour for the tree where the check searches for the next element in the ordering. Meanwhile, the list implementation is usually a $O(1)$ dereference and compare, but sometimes needs to iterate further when the starting capability has copies.

For both implementations, the check for copies shown in Figure 4.5 uses one forward and back iteration. Because iteration in the tree has two modes,

performing two iterations and short circuiting if the first suffices results in three modes visible in the figure.

The retype loop at the root of the capability type tree means that when searching for ancestors by iterating backwards, there can always be an ancestor earlier in the ordering with a bigger size. This results in an $O(n)$ complexity for the linked-list implementation as can be seen in Figure 4.6. The presence of an ancestors allows the algorithm to terminate early, but does not reduce the $O(n)$ complexity, and is not the common case. The tree implementation meanwhile performs first a search for a predecessor that is not a copy, and when that fails, a range query as described in section 4.3. Successes of the first case can be seen as a line of outliers below the dominating case that uses a range query.

Finally, querying for the smallest capability covering a given address is an example of a single range query. This operation was not implemented for the linked-list MDB, but would have required a linear scan through the database. The results for the tree-based MDB are shown in Figure 4.7.

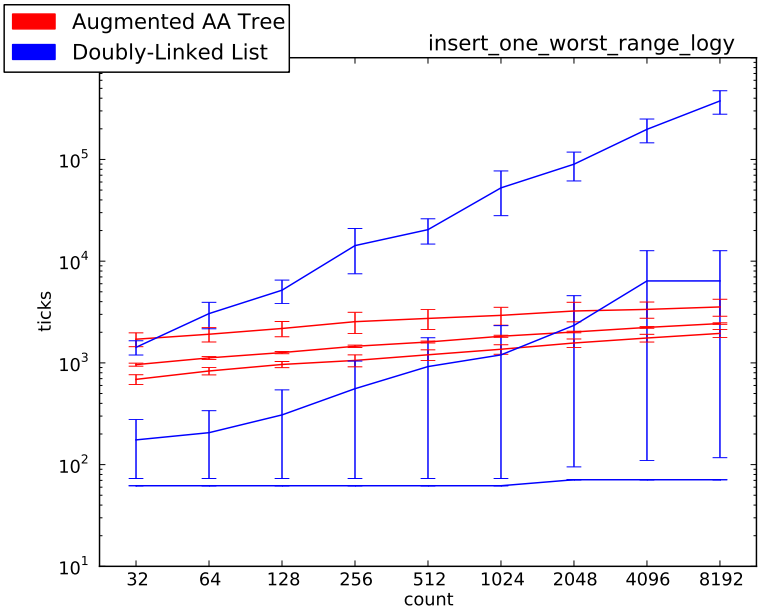

Figure 4.1: Time in ticks to insert an element into an MDB already containing *count* elements. The plot shows the top, median and bottom 10% of measurements for each tree size.

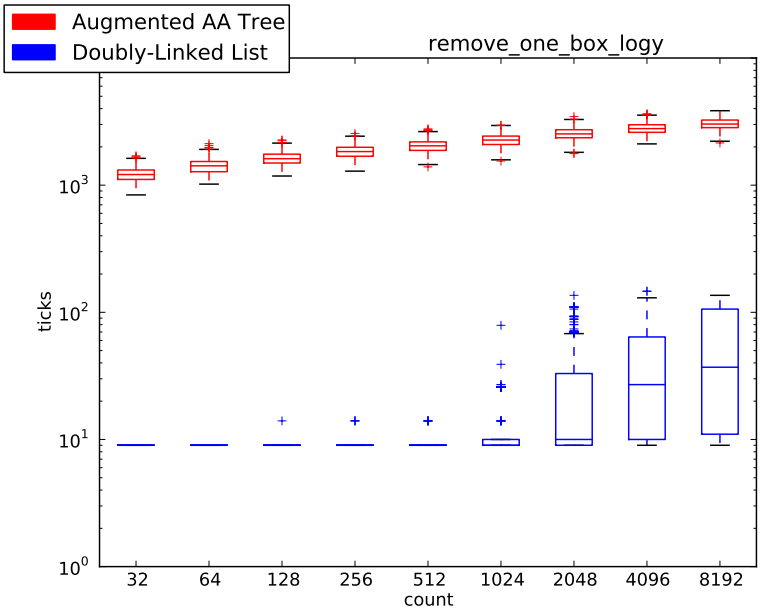

Figure 4.2: Time in ticks to remove an element from an MDB containing *count* elements.

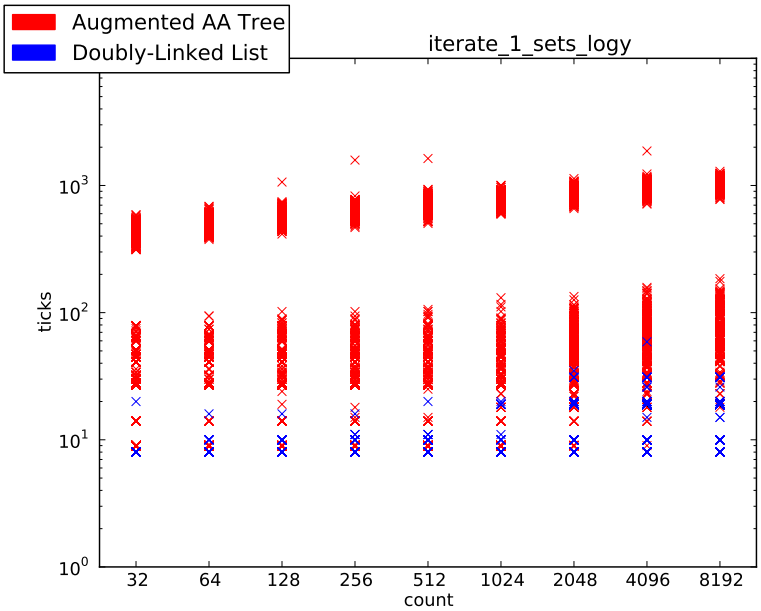

Figure 4.3: Time in ticks to iterate forwards once in an MDB containing *count* elements.

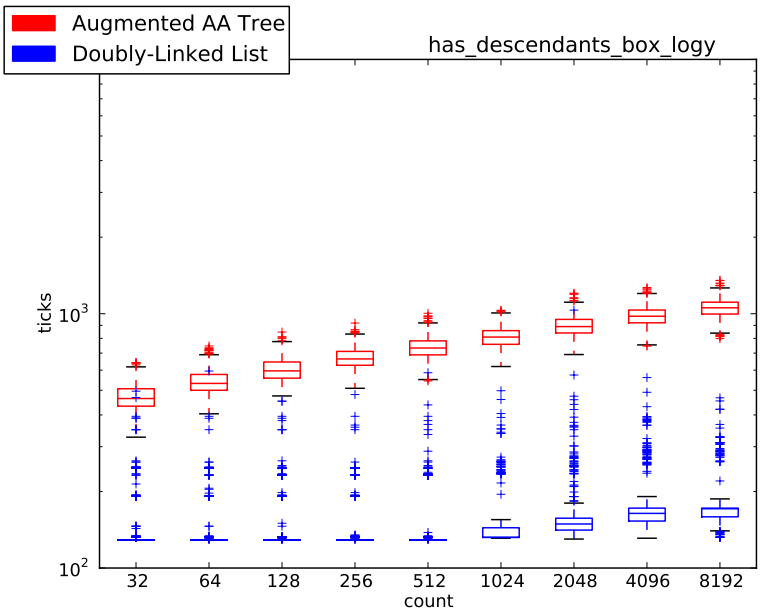

Figure 4.4: Time in ticks to query if an element has descendants in an MDB containing *count* elements.

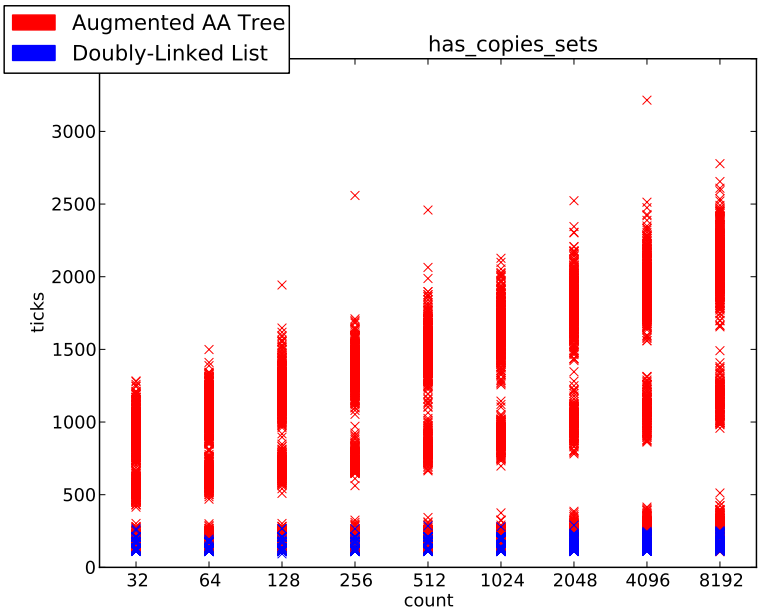

Figure 4.5: Time in ticks to query if an element has copies in an MDB containing *count* elements.

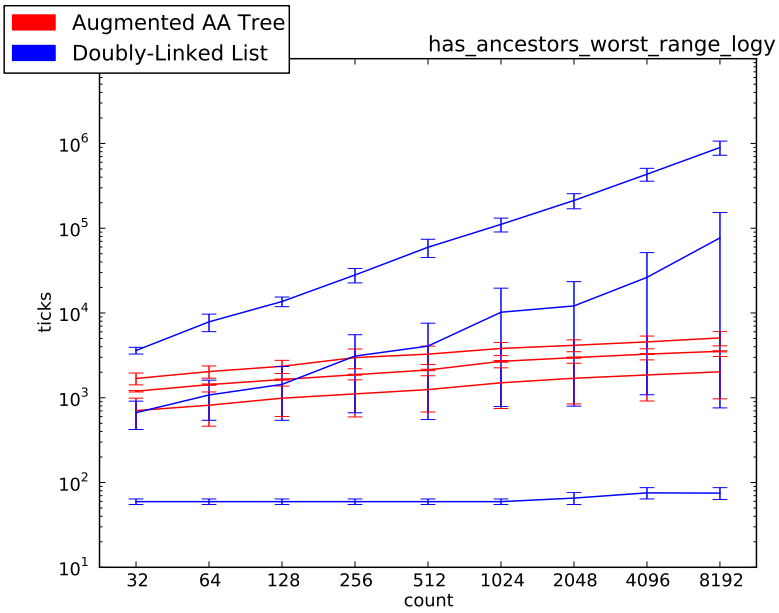

Figure 4.6: Time in ticks to query if an element has ancestors in an MDB containing *count* elements. The plot shows the top, median and bottom 10% of measurements for each tree size.

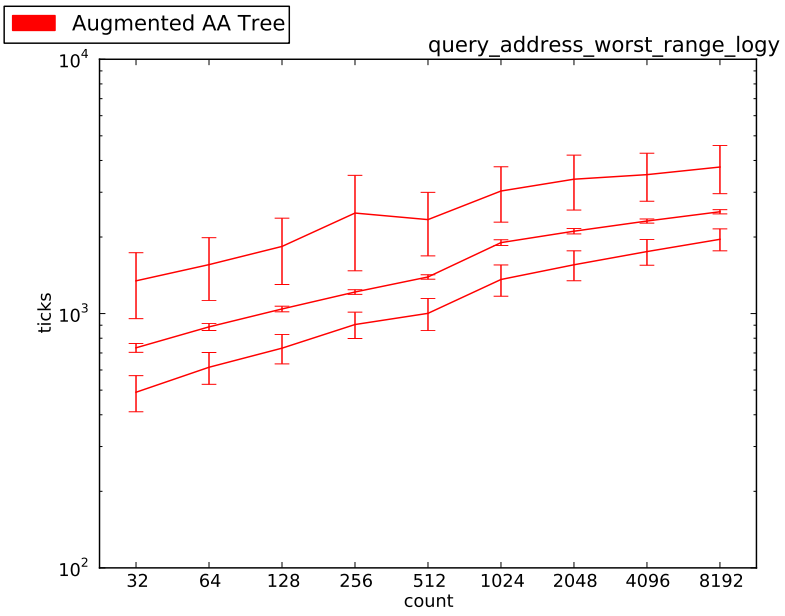

Figure 4.7: Time in ticks to find the smallest element containing an address in an MDB containing *count* elements. The plot shows the top, median and bottom 10% of measurements for each tree size.

Chapter 5

Conclusion & Future Work

By designing a distributed capability system for use in Barrelfish, we have seen how the multi-kernel model affects the structure of a capability system for managing operating system resources. Starting with a local capability system derived from the system used by the seL4 microkernel, we have demonstrated a design similar to that found in distributed object systems. With the ownership property introduced by this design, we are able to synchronize operations; not just the operations used for manipulation of capability slots, but also operations on other objects types in the system.

The ability to apply ideas from distributed object systems to multi-kernel capability systems indicates that this model is well-suited, although care must be taken when implementing operations that affect many capabilities in the system. The limited set of operations greatly simplifies this task, as it becomes possible to cross-compare all operations for interference.

In distributed object systems, care must also must be taken to ensure reliability of remote invocations. In Barrelfish, partial failures like undelivered messages or unresponsive nodes are not currently supported, so all messages are expected to be delivered eventually, and we have therefore not explored the handling of such failure types. However, when trying to handle such scenarios, the limited set of operations might help us again, e.g. if we are able to engineer all operations to be sufficiently idempotent that they may be rerun under some kinds of failures.

With the non-sharing nature of the chosen multi-kernel model, the database of capability relations derived from seL4's Capability Derivation Tree becomes split, with only a partial views of the set of capabilities available on each core. The extensions required to the database to perform queries based on capabilities not present in the local database have lead us to implement an alternative tree-based datastructure, the initial version of which has shown significant performance degredation for previous operations as $O(1)$ algorithms have been replaced with $O(\log n)$ alternatives. Merging the datastructures would return the performance close to the previous datastructure at the cost of an increase in capability slot size as information from both structures must

coexist within each capability slot.

The introduction of "foreign" capabilities that act as proxies allows capabilities to be used by nodes without direct read/write access to the objects represented by those capabilities. While this enables spanning the capability system across nodes that do not share a memory system, it raises the questions of what invocations are useful when the object cannot be accessed. For example, it may not be possible to map a remote frame capability into virtual address spaces on some nodes.

Evaluating the performance of an implementation of our design (an implementation was begun but not completed for this thesis) would allow further insight into the scalability of this system, for example with regard to the amount of lock contention. A count of total cross-core operations performed in common scenarios for multi-threaded applications might indicate which operations might need strong optimizations, while measurement of message load caused by distributed operations could indicate where optimization of multi-cast message routing is necessary.

Appendix A

Hamlet Examples

```
/** We can define some constants using the "define"
    construct
**/
/* Size of CNode entry: */
define cte_size 7;

/** The capabilities of the whole system are listed
    thereafter. The minimal definition consists of a
    name and an empty body.
**/

cap Null is_never_copy {
    /* Null/invalid object type */
};

cap PhysAddr from_self {
    /* Physical address range */

    /**
      For a populated cap, we need to give the type and
      name of each of its fields, such as:
      "genpaddr base;"

      In order to implement various comparisons, we
      need to specify an address and size for each type
      that is backed by memory. The size may be
      specified directly with "size" or as "size_bits".

      Additional equality fields can be specified with
      an "eq" prefix, as in:
      "eq genpaddr base;"
```

```
    **/

    address genpaddr base; /* Base address of
                              untyped region */
    size_bits uint8 bits;  /* Address bits that
                              untyped region bears */

};

cap RAM from PhysAddr from_self {
    /* RAM memory object */

    address genpaddr base; /* Base address of untyped
                              region */
    size_bits uint8 bits;  /* Address bits that untyped
                              region bears */

};

cap CNode from RAM {
    /* CNode table, stores further capabilities */

    lpaddr cnode;          /* Base address of CNode */
    uint8 bits;            /* Number of bits this CNode
                              resolves */
    caprights rightsmask;
    uint8 guard_size;      /* Number of bits in guard */
    caddr guard;           /* Bitmask already resolved
                              when reaching this CNode */

    /**
      Address and size may also be specified with some
      very limited expressions.
    **/

    address { cnode };
    size_bits { bits + cte_size };
};

cap Frame from RAM from_self {
    /* Mappable memory frame */

    address genpaddr base;
    size_bits uint8 bits;
};

cap DevFrame from PhysAddr from_self {
    /* Mappable device frame */

    address genpaddr base;
```

```
    size_bits uint8 bits;
};

cap Kernel is_always_copy {
    /* Capability to a kernel */
};
```

Appendix B

TLA+ Specification

To illustrate the semantics of the system, we have translated the behaviour specified in chapter 2 into a TLA+ specification, shown on the next pages. The system specified is *not* a distributed system, modelling only one ongoing operation at a time, and serves only to show that the indicated behaviour fulfils the invariants and thus any system implementing the same behaviour does so to.

The specification was model-checked with TLC, a checker for TLA+ specifications that enumerates all distinct states using a BFS of state space. To limit the number of states that must be explored by TLC, a very small system was used as model, with parameters indicated in Table B.1. Figure B.1 shows that this model was sufficient to execute all actions.

CapTypes $\triangleq$ {“Null”, “Mem”, “CNode”}	Addressable(t) $\triangleq$ $t \in$ {“CNode”, “Mem”}
Null $\triangleq$ “Null”	Moveable(t) $\triangleq$ $t =$ “Mem”
RetypeSource(t) $\triangleq$ IF $t =$ “CNode” THEN “Mem” ELSE “”	Splittable(t) $\triangleq$ $t =$ “Mem”
Mem $\triangleq$ “Mem”	NumCores $\triangleq$ 2
PSpaceSize $\triangleq$ 3	NumSlots $\triangleq$ 4
	NumOps $\triangleq$ 1

Table B.1: The checked model

```
The coverage statistics at 2012-04-28 21:27:11
  line 159, col 5 to line 159, col 66 of module globalsys: 1369376
  line 250, col 8 to line 254, col 45 of module globalsys: 149183136
  line 255, col 18 to line 255, col 22 of module globalsys: 149183136
  line 260, col 5 to line 260, col 56 of module globalsys: 148875552
  line 268, col 29 to line 268, col 33 of module globalsys: 4817280
  line 277, col 29 to line 277, col 33 of module globalsys: 130066560
  line 280, col 29 to line 280, col 33 of module globalsys: 10917824
  line 293, col 29 to line 293, col 33 of module globalsys: 104304
  line 307, col 19 to line 310, col 42 of module globalsys: 234624
  line 317, col 20 to line 323, col 46 of module globalsys: 56960
  line 331, col 29 to line 331, col 33 of module globalsys: 104304
  line 334, col 19 to line 342, col 45 of module globalsys: 1204320
  line 359, col 9 to line 359, col 54 of module globalsys: 67186558464
  line 360, col 18 to line 360, col 22 of module globalsys: 67186558464
End of statistics.
67484617608 states generated, 111426304 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 42.
Finished.
```

Figure B.1: Coverage of actions.

```
1  ──────────────────── MODULE globalsys ────────────────────
2  EXTENDS Naturals

4    This module illustrates the invariants specified for the capability system
5    using a specification that matches the pseudocode in chapter 2.

7  ──────────────────────────────────────────────────────────

9    Requirements for a capability type system:

11   All capability types, and the null type specially identified
12 CONSTANT CapTypes
13 CONSTANT Null

15   Predicates for retype relations and mutability of ownership.
16 CONSTANTS RetypeSource(_), Moveable(_)

18   Null must be a valid type.
19 ASSUME Null ∈ CapTypes

21   Use an implicit "NoType" value to indicate a type has no parent.
22 NoType ≜ RetypeSource(Null)
23 ASSUME NoType ∉ CapTypes

25   With NoType defined, type requirements for RetypeSource become possible:
26 FromTypes ≜ (CapTypes ∪ {NoType}) \ {Null}  Cannot retype from Null
27 ASSUME ∀ t ∈ CapTypes : RetypeSource(t) ∈ (FromTypes \ {t})

29 ──────────────────────────────────────────────────────────

31   Next, requirements for addressable capabilities:

33   A basic memory type. This constant is only required for setting up the inital state
34 CONSTANT Mem

36   Additioal predicates for addressable capabilities
37 CONSTANTS Splittable(_), Addressable(_)

39   Addressable and Moveable must be defined for all types
40 ASSUME ∀ t ∈ CapTypes :
41       ∧   Addressable(t) ∈ BOOLEAN
42       ∧   Moveable(t) ∈ BOOLEAN

44   Null is not addressable, while Mem is
45 ASSUME Addressable(Null) = FALSE
46 ASSUME Addressable(Mem) = TRUE

48   Addressability applies to a whole tree of the type forest.
49 ASSUME ∀ t ∈ CapTypes :
50     LET p ≜ RetypeSource(t)
```

```
51      IN    p ≠ NoType ⇒ (Addressable(t) ≡ Addressable(p))

53    Only addressable caps can be split.
54  ASSUME ∀ t ∈ CapTypes :
55        ∧    Splittable(t) ∈ BOOLEAN
56        ∧    (¬Addressable(t)) ⇒ (¬Splittable(t))
57        ∧    Splittable(t) ⇒ Addressable(t)

59  Ancestors(t) ≜
60      IF RetypeSource(t) = NoType
61       THEN {}
62       ELSE  CHOOSE s ∈ SUBSET CapTypes :
63           ∧ RetypeSource(t) ∈ s
64           ∧ t ∉ s
65           ∧ ∀ parent ∈ s : ∃ desc ∈ (s ∪ {t}) : RetypeSource(desc) = parent

67 ├──────────────────────────────────────────────────────────────────────┤

69    Requirements and definitions for PSpace:

71  CONSTANT PSpaceSize
72  ASSUME PSpaceSize ∈ Nat
73  MaxPAddr ≜ PSpaceSize − 1
74  PSpace ≜ 0 .. MaxPAddr   PSpace should not have to be contiguous, but this
75                                simplifies the sepcification.

77    The set of all contiguous address ranges that are subsets of PSpace
78  PRanges ≜ {r ∈ SUBSET PSpace :
79                ∃ b ∈ PSpace, s ∈ 1 .. PSpaceSize : ∀ a ∈ r : a ∈ (b .. (b + s − 1))}

81    Regions are ranges of PSpace given by a base and size
82  Regions ≜ [base : PSpace, size : 1 .. (MaxPAddr + 1)]
83  NoRegion ≜ CHOOSE r : r ∉ Regions
84  RegionAddrs(r) ≜ r.base .. (r.base + r.size − 1)

86 ├──────────────────────────────────────────────────────────────────────┤

      Cores in system. As with PSpace, this should not have to be contiguous, but making it so
      simplifies things.

91  CONSTANT NumCores
92  ASSUME NumCores ∈ Nat
93  Cores ≜ 0 .. (NumCores − 1)

95 ├──────────────────────────────────────────────────────────────────────┤

97  CONSTANTS NumSlots, NumOps

99  VARIABLES slots, operations
```

```
101 ├──────────────────────────────────────────────────────────────────────┤

103 Caps ≜ [type : CapTypes, region : Regions ∪ {NoRegion}]
104 NullCap ≜ [type ↦ Null, region ↦ NoRegion]
105 ASSUME NullCap ∈ Caps

107 IsCapCopy(cap1, cap2) ≜
108     ∧ cap1.type = cap2.type
109     ∧ cap1.region = cap2.region

111 ├──────────────────────────────────────────────────────────────────────┤

113 Retyped(cap, region, type) ≜
114     [type ↦ type,
115      region ↦ region]

117 IsAncestor(child, ancestor) ≜
118     ∨ ∧ child.type = ancestor.type
119       ∧ Addressable(child.type)
120       ∧ Splittable(child.type)
121       ∧ child.region ≠ ancestor.region
122       ∧ RegionAddrs(child.region) ⊆ RegionAddrs(ancestor.region)
123     ∨ ∧ ancestor.type ∈ Ancestors(child.type)
124       ∧ ¬Addressable(child.type)
125     ∨ ∧ ancestor.type ∈ Ancestors(child.type)
126       ∧ Addressable(child.type)
127       ∧ RegionAddrs(child.region) ⊆ RegionAddrs(ancestor.region)

129 CanRetype(cap, region, type) ≜
130     ∧ type ∈ CapTypes
131     ∧ region ≠ NoRegion
132       ⇒ RegionAddrs(region) ⊆ PSpace
133     ∧ ∨ RetypeSource(type) = cap.type
134       ∨ cap.type = type ∧ Splittable(type)
135     ∧ IsAncestor(Retyped(cap, region, type), cap)
136     ∧ ∀ s ∈ DOMAIN slots :
137         LET scap ≜ slots[s].cap
138             rcap ≜ Retyped(cap, region, type)
139         IN  ∨ scap.type = Null
140             ∨ IsAncestor(rcap, scap)
141             ∨ (Addressable(rcap.type) ∧ Addressable(scap.type))
142               ⇒ (RegionAddrs(rcap.region) ∩ RegionAddrs(scap.region)) = {}

144 ├──────────────────────────────────────────────────────────────────────┤

146 NoOwner ≜ CHOOSE o : o ∉ Cores

148 SlotIds ≜ 0 .. (NumSlots − 1)
```

```
150 Slots ≜ [cap : Caps, owner : (Cores ∪ {NoOwner}), location : Cores]

152 SlotWithCap(slot, cap, owner) ≜
153     [slot EXCEPT !.cap = cap, !.owner = owner]

155 SlotWithNullCap(slot) ≜
156     SlotWithCap(slot, NullCap, NoOwner)

158 SetSlot(slotid, cap, owner) ≜
159     slots′ = [slots EXCEPT ![slotid] = SlotWithCap(@, cap, owner)]

161 CopySlot(destid, srcid) ≜
162     SetSlot(destid, slots[srcid].cap, slots[srcid].owner)

164 IsSlotCopy(sid1, sid2) ≜
165     IsCapCopy(slots[sid1].cap, slots[sid2].cap)

167 ClearSlot(sid) ≜
168     SetSlot(sid, NullCap, NoOwner)

170 SlotInvariants ≜
171       Type correctness of slot array
172     ∧ DOMAIN slots ⊆ SlotIds
173     ∧ ∀ sid ∈ DOMAIN slots : slots[sid] ∈ Slots
174       Only Null caps may not have an owner
175     ∧ ∀ sid ∈ DOMAIN slots : slots[sid].owner = NoOwner
176                                 ≡ slots[sid].cap.type = Null

178 SingleLocationProperty ≜
179       A slots location must never change
180     ∧ ∀ sid ∈ DOMAIN slots : slots[sid].location = slots′[sid].location

182 OwnershipInvariants ≜
183       All copies of a cap have the same owner
184     ∀ sid1 ∈ DOMAIN slots, sid2 ∈ DOMAIN slots :
185       IsSlotCopy(sid1, sid2) ⇒ slots[sid1].owner = slots[sid2].owner

187 ImmutabilityProperty ≜
188       A non-null cannot be modified without first being deleted
189     ∀ sid ∈ DOMAIN slots :
190       ( ∧ slots[sid].cap.type ≠ Null
191         ∧ slots′[sid].cap.type ≠ Null)
192         ⇒ IsCapCopy(slots′[sid].cap, slots[sid].cap)

194 ├──────────────────────────────────────────────────────────────────────────┤

196 CopyReq ≜ [name : {"copy"}, src : SlotIds, dest : SlotIds]
197 RetypeReq ≜ [name : {"retype"}, src : SlotIds, region : Regions, type : CapTypes, dest : SlotIds]
198 DeleteReq ≜ [name : {"delete"}, target : SlotIds]
```

```
199 RevokeReq ≜ [name : {"revoke"}, target : SlotIds]
200 RequestTypes ≜ CopyReq ∪ RetypeReq ∪ DeleteReq ∪ RevokeReq

202 CopyOp ≜ [name : {"copy"}, src : Caps, owner : (Cores ∪ {NoOwner}), dest : SlotIds]
203 RetypeOp ≜ [name : {"retype"}, src : Caps, region : Regions, type : CapTypes, dest : SlotIds]
204 DeleteOp ≜ [name : {"delete"}, target : SlotIds]
205 RevokeOp ≜ [name : {"revoke"}, target : SlotIds, target_cap : Caps]
206 OperationTypes ≜ CopyOp ∪ RetypeOp ∪ DeleteOp ∪ RevokeOp

208 OperationStates ≜ {"running", "failed", "succeeded"}

210 NewRequests ≜ [req : RequestTypes, launched : {FALSE}]
211 LaunchedRequests ≜ [req : RequestTypes, launched : {TRUE}, op : OperationTypes, state : OperationStates]
212 OperationComplete(o) ≜ o.state ∈ {"failed", "succeeded"}

214 Operations ≜ NewRequests ∪ LaunchedRequests

216 OperationIds ≜ 0 .. (NumOps − 1)

218 OperationInvariants ≜
219     ∧ DOMAIN operations ⊆ OperationIds
220     ∧ ∀ o ∈ DOMAIN operations : operations[o] ∈ Operations

222 ├──────────────────────────────────────────────────────────────────────────────┤

224 CanStart(req) ≜
225     CASE req.name = "copy" →
226             ∧ req.src ∈ DOMAIN slots
227             ∧ req.dest ∈ DOMAIN slots
228       □req.name = "retype" →
229         ∧ req.src ∈ DOMAIN slots
230         ∧ req.dest ∈ DOMAIN slots
231       □req.name = "delete" →
232         ∧ req.target ∈ DOMAIN slots
233       □req.name = "revoke" →
234         ∧ req.target ∈ DOMAIN slots

236 MkRequestOp(req) ≜
237     CASE req.name = "copy"   → [name ↦ req.name,
238                                 src ↦ slots[req.src].cap,
239                                 owner ↦ slots[req.src].owner,
240                                 dest ↦ req.dest]
241       □req.name   = "retype" → [req EXCEPT !.src = slots[@].cap]
242       □req.name   = "delete" → req
243       □req.name   = "revoke" → [name ↦ req.name,
244                                 target ↦ req.target,
245                                 target_cap ↦ slots[req.target].cap]
```

```
StartOp(oid) ≜
    ∧ ¬operations[oid].launched
    ∧ CanStart(operations[oid].req)
    ∧ operations' = [operations EXCEPT ![oid] = [
                        req ↦ @.req,
                        launched ↦ TRUE,
                        op ↦ MkRequestOp(@.req),
                        state ↦ "running"]]
    ∧ UNCHANGED slots

──────────────────────────────────────────────────────────────────────

SetOpState(o, state) ≜
    operations' = [operations EXCEPT ![o].state = state]

FailOp(o) ≜ SetOpState(o, "failed")
SucceedOp(o) ≜ SetOpState(o, "succeeded")

RunCopy(o) ≜
    LET op ≜ operations[o].op
    IN  CASE slots[op.dest].cap.type ≠ Null
              → ∧ UNCHANGED slots
                ∧ FailOp(o)
          □OTHER
              → ∧ SetSlot(op.dest, op.src, op.owner)
                ∧ SucceedOp(o)

RunRetype(o) ≜
    LET op ≜ operations[o].op
    IN  CASE slots[op.dest].cap.type ≠ Null
              → ∧ UNCHANGED slots
                ∧ FailOp(o)
          □¬CanRetype(op.src, op.region, op.type)
              → ∧ UNCHANGED slots
                ∧ FailOp(o)
          □OTHER
              → ∧ LET retyped ≜ Retyped(op.src, op.region, op.type)
                    IN  SetSlot(op.dest, retyped, slots[op.dest].location)
                ∧ SucceedOp(o)

RunDelete(o) ≜
    LET op ≜ operations[o].op
        slotid ≜ op.target
        slot ≜ slots[op.target]
    IN  CASE slot.cap.type = Null
                Deleting Null slots is OK and a no-op
              → ∧ UNCHANGED slots
```

```
                    ∧ SucceedOp(o)
          □slot.location ≠ slot.owner
             Non-owned, just delete
             → ∧ ClearSlot(slotid)
               ∧ SucceedOp(o)
          □(∃ s ∈ DOMAIN slots : ∧ s ≠ slotid
                                 ∧ slots[s].location = slot.location
                                 ∧ IsSlotCopy(s, slotid))
             Have copies on same core, just delete
             → ∧ ClearSlot(slotid)
               ∧ SucceedOp(o)
          □¬Moveable(slot.cap.type)
             Cannot move, delete all copies
             → ∧ slots′ = [s ∈ DOMAIN slots ↦
                          IF IsSlotCopy(s, slotid)
                           THEN SlotWithNullCap(slots[s])
                           ELSE slots[s]]
               ∧ SucceedOp(o)
          □OTHER
             Migrate ownership and delete
             → ∃ s ∈ DOMAIN slots :
                 ∧ s ≠ slotid
                 ∧ IsSlotCopy(s, slotid)
                 ∧ slots′ = [c ∈ DOMAIN slots ↦
                              CASE c = slotid
                                   → SlotWithNullCap(slot)
                                □IsSlotCopy(c, slotid)
                                   → [slots[c] EXCEPT !.owner = slots[s].location]
                                □OTHER
                                   → slots[c]]
                 ∧ SucceedOp(o)

RunRevoke(o) ≜
   LET op ≜ operations[o].op
       slotid ≜ op.target
       slot ≜ slots[op.target]
   IN  CASE slot.cap.type = Null
            → ∧ UNCHANGED slots
              ∧ FailOp(o)
         □OTHER
            → ∧ slots′ = [s ∈ DOMAIN slots ↦
                          CASE s = slotid
                               → slots[s]
                            □IsSlotCopy(s, slotid)
                               → SlotWithNullCap(slots[s])
```

```
                                    □IsAncestor(slots[s].cap, slot.cap)
                                      → SlotWithNullCap(slots[s])
                                    □OTHER
                                      → slots[s]]
                  ∧ SucceedOp(o)

CompleteOp(o) ≜
    ∧ operations[o].launched
    ∧ operations[o].state = "running"
    ∧ LET op ≜ operations[o].op
          name ≜ op.name
      IN  CASE name = "copy" → RunCopy(o)
            □name   = "retype" → RunRetype(o)
            □name   = "delete" → RunDelete(o)
            □name   = "revoke" → RunRevoke(o)

ResetOp(o) ≜
    ∧   operations[o].launched
    ∧   operations[o].state ∈ {"failed", "succeeded"}
    ∧   ∃ newop ∈ NewRequests :
          operations' = [operations EXCEPT ![o] = newop]
    ∧   UNCHANGED slots

──────────────────────────────────────────────────────────────────────

TypeInvariant ≜
    ∧ SlotInvariants
    ∧ OwnershipInvariants
    ∧ OperationInvariants

Init ≜
    ∧ slots = [s ∈ 0 .. (NumSlots − 1) ↦
                IF s = 0
                 THEN [cap ↦ [type ↦ Mem,
                              region ↦ [base ↦ 0,
                                        size ↦ MaxPAddr + 1]],
                       owner ↦ 0,
                       location ↦ s%NumCores]
                 ELSE [cap ↦ NullCap,
                       owner ↦ NoOwner,
                       location ↦ s%NumCores]]
    ∧ operations ∈ [0 .. (NumOps − 1) → NewRequests]

Next ≜  ∧ ∃ o ∈ DOMAIN operations :
            ∨ StartOp(o)
            ∨ CompleteOp(o)
            ∨ ResetOp(o)
```

386 $\land$ TRUE $\lor$ *SingleLocationProperty*
387 $\land$ TRUE $\lor$ *ImmutabilityProperty*

389 $Spec \triangleq Init \land \Box[Next]_{\langle slots,\, operations \rangle}$

391 ⊢———————————————————————————————————————⊣

393 THEOREM $Spec \Rightarrow \Box\, TypeInvariant$

395 └———————————————————————————————————————┘

Bibliography

[1] Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Timothy L. Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 29–44. ACM, 2009.

[2] 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*, pages 319–327, San Jose, California, October 4–7, 1994. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[3] Maurice D. Castro, Ronald D. Pose, and Carlo Kopp. Password-capabilities and the Walnut Kernel. *The Computer Journal*, 51(5):595–607, September 2008.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2 edition, September 2001.

[5] Simon Gerber. Virtual memory in a multikernel. Master's thesis, ETH Zürich, May 2012.

[6] Steven M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the third symposium on operating systems design and implementation*, OSDI '99, pages 73–86, Berkeley, CA, USA, 1999. USENIX Association.

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[8] Donald E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.

[9] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 166–176, Oakland, CA, May 2000. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[10] Akhilesh Singhania and Ihor Kuz. Capability management in Barrelfish. Barrelfish Technical Note 13, Systems Group, ETH Zurich, 2011.

[11] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. In USENIX, editor, *Proceedings of the Second USENIX Conference on Object-Oriented Technologies and Systems (COOTS), June 17–21, 1996, Toronto, Canada*, pages 219–231, pub-USENIX:adr, 1996. USENIX.