SemperOS: A Distributed Capability System

Matthias Hille[†] Nils Asmussen[†] [*] Pramod Bhatotia[‡] Hermann Härtig[†] [*]
[†]*Technische Universität Dresden* [‡]*The University of Edinburgh* [*] *Barkhausen Institut*

Abstract

Capabilities provide an efficient and secure mechanism for fine-grained resource management and protection. However, as the modern hardware architectures continue to evolve with large numbers of non-coherent and heterogeneous cores, we focus on the following research question: *can capability systems scale to modern hardware architectures?*

In this work, we present a scalable capability system to drive future systems with many non-coherent heterogeneous cores. More specifically, we have designed a distributed capability system based on a HW/SW co-designed capability system. We analyzed the pitfalls of distributed capability operations running concurrently and built the protocols in accordance with the insights. We have incorporated these distributed capability management protocols in a new microkernel-based OS called SemperOS. Our OS operates the system by means of multiple microkernels, which employ distributed capabilities to provide an efficient and secure mechanism for fine-grained access to system resources. In the evaluation we investigated the scalability of our algorithms and run applications (Nginx, LevelDB, SQLite, PostMark, etc.), which are heavily dependent on the OS services of SemperOS. The results indicate that there is no inherent scalability limitation for capability systems. Our evaluation shows that we achieve a parallel efficiency of 70% to 78% when examining a system with 576 cores executing 512 application instances while using 11% of the system's cores for OS services.

1 Introduction

Capabilities are unforgeable tokens of authority granting rights to resources in the system. They can be selectively delegated between constrained programs for implementing the principle of least authority [48]. Due to their ability of fine-grained resource management and protection, capabilities appear to be a particularly good fit for future hardware architectures, which envision byte granular memory access to large memories (NVRAM) from a large numbers of cores (e.g. The Machine [31], Enzian [18]). Thereby, capability-based systems have received renewed attention recently to provide an efficient and secure mechanism for resource management in modern hardware architectures [5, 24, 30, 36, 44, 64, 67].

Today the main improvements in compute capacity are achieved by either adding more cores or integrating accelerators into the system. However, the increasing core counts exacerbate the hardware complexity required for global cache coherence. While on-chip cache coherence is likely to remain a feature of future hardware architectures [45], we see characteristics of distributed systems added to the hardware by giving up on global cache coherence across a whole machine [6, 28]. Additionally, various kinds of accelerators are added like the Xeon Phi Processor, the Matrix-2000 accelerator, GPUs, FPGAs, or ASICs, which are used in numerous application fields [4, 21, 32, 34, 43, 60]. These components also contribute to the number of resources an OS has to manage.

In this work, we focus on capability-based systems and how their ability to implement fine-grained access control combines with large systems. In particular, we consider three types of capability systems: L4 [30], CHERI [67] (considered for The Machine [31]), and M^3 [5] (see Section 2.1). For all these capability types it is not clear whether they will scale to modern hardware architectures since the scalability of capability systems has never been studied before. Also existing capability schemes cannot be turned into distributed schemes easily since they either rely on centralized knowledge, cache-coherent architectures, or are missing important features like revocation.

Independent of the choice which capability system to use, scaling these systems calls for two basic mechanisms to be fast. First, it implies a way of concurrently updating access rights to enable fast decentralized resource sharing. This means fast *obtaining* or *delegating* of capabilities, which acquires or hands out access rights to the resources behind the capabilities. The other performance-critical mechanism is the revocation of capabilities. *Revoking* the access rights should be possible within a reasonable amount of time and with minimal overhead. The scalability of this operation is tightly coupled to the enforcement mechanism, e.g. when using L4 capabilities the TLB shootdown can be a scalability bottleneck.

We base our system on a hardware/software co-designed capability system (M^3). More specifically, we propose a scalable distributed capability mechanism by building a multikernel OS based on the M^3 capability system. We present a detailed analysis of possible complications in distributed capability systems caused by concurrent updates. Based on this investigation we describe the algorithms, which we implemented in our prototype OS—the SEMPEROS multikernel.

Our OS divides the system into groups, with each of them being managed by an independent kernel. These independently managed groups resemble islands with locally managed resources and capabilities. Kernels communicate via messages to enable interaction across groups. To quickly find objects across the whole system—a crucial prerequisite for our capability scheme—we introduce an efficient addressing scheme.

Our system design aims at future hardware, which might connect hundreds of processing elements to form a powerful rack-scale system [28]. To be able to experiment with such systems, we use the gem5 simulator [14] Our evaluation focuses on the performance of the kernel, where we showcase the scalability of our algorithms by using microbenchmarks as well as OS-intensive real-world applications. We describe trade-offs in resource distribution between applications and the OS to determine a suitable configuration for a specific application. We found that our OS can operate a system which hosts 512 parallel running application instances with a parallel efficiency of 70% to 78% while dedicating 11% of the system to the OS and its services.

To summarize, our contributions are as follows.

- We propose a HW/SW co-designed distributed capability system to drive future systems. Our capability system extends M^3 capabilities to support a large number of non-cache-coherent heterogeneous cores.
- We implemented a new microkernel-based OS called SEMPEROS that operates the system by employing multiple microkernels and incorporates distributed capability management protocols.
- We evaluated the distributed capability management protocols by implementing the HW/SW co-design for SEMPEROS in the gem5 simulator [14] to run real applications: Nginx, SQLite, PostMark, and LevelDB.

2 Background

We first assess existing capability systems and explain the basic principles of M^3, which is the foundation of our work.

2.1 Capability Systems

The term capability was first used by Dennis and van Horn [20] to describe a pointer to a resource, which provides the owner access to the resource. There are three basic types of capabilities: (1) partitioned capabilities, which have been employed in multiple OSes such as KeyKOS [27], EROS [56] and various L4 microkernels [30, 36, 40, 62], (2) instruction set architecture (ISA) capabilities, as implemented by the CAP computer [49] and recently revived by CHERI [67] and (3) sparse capabilities which are deployed in the password-capability system of the Monash University [2] and in Amoeba [63].

	L4	M^3	CHERI
Scope	Coherence Dom.	Machine	Address space
Enforcement	MMU / Kernel	DTU / Kernel	CHERI co-proc.
Limitation	Coherence Dom.	Core count	no revoke

Table 1: Classification of capability types.

Capabilities can be shared to exchange access rights. ISA capabilities and sparse capabilities can be shared without involving the kernel since their validity is either ensured by hardware or checked by the kernel using a cryptographic one-way function in the moment they are used. In contrast, sharing of partitioned capabilities is observed by the kernel.

To analyze capability systems regarding their scalability, we inspect their enforcement mechanism, their scope, and their limitation in Table 1. The three categories of capability systems in Table 1 represent a relevant subset of capability systems for the scope of this work: (1) L4 capabilities which are partitioned capabilities employed in L4 μ-kernels [30, 36, 40, 62], (2) M^3 capabilities which are a special form of partitioned capabilities involving a different enforcement mechanism explained in the following, and lastly (3) CHERI capabilities which are ISA capabilities implemented by the CHERI processor [64, 67].

L4 capabilities utilize the memory management unit (MMU) of the processor to restrict a program's memory access. Access to other resources like communication channels or process control are enforced by the kernel. Since L4 is built for cache coherent machines both the scope of a capability and the current limitation is a coherence domain.

In contrast, M^3 introduces a hardware component called data transfer unit (DTU) which provides message passing between processing elements and a form of remote direct memory access. Consequently, memory access and communication channels are enforced by the DTU and access to other system resources by the kernel. The DTU is the only possibility for a core to interact with other components. Hence it can be used to control a core's accesses to system resources via NoC-level isolation. (We will give a more detailed explanation of M^{3}capabilities in the following section.) Importantly, M^3 capabilities are valid within a whole machine spanning multiple coherence domains. However, M^3 is currently limited by using a single kernel core to control multiple application cores.

Lastly, the ISA capabilities of the CHERI system are enforced by a co-processor. CHERI capabilities contain a description of the resource, i.e., memory they point to. This information is used by the co-processor to determine the validity of accesses. The scope of a CHERI capability is an address space. Thus, such a system typically uses one address space for multiple applications. However, CHERI does not support revocation and therefore does not have the problem we are solving.

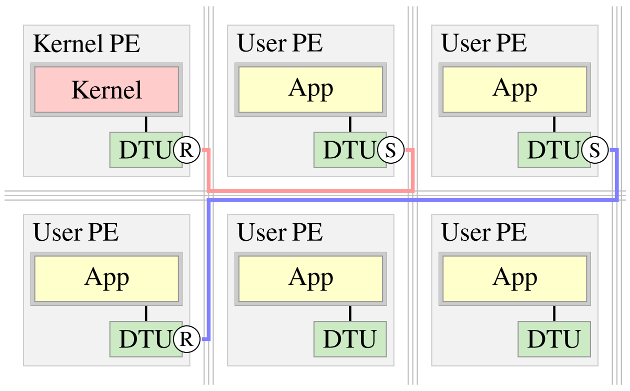

Figure 1: System architecture of M^3. Each processing element (PE) has a data transfer unit (DTU) connecting them to the network-on-chip. DTUs are configured by the kernel PE.

For both L4-style and M^3-style capabilities, scaling to larger systems and maintaining consistency demands the extension to multiple kernels and their coordination. For L4-style systems, multiple kernels are required to scale beyond coherence domains. For M^3-style systems, multiple kernels are required to scale to large core counts.

2.2 M^3: HW/SW Co-designed Capabilities

To accommodate for the hardware trends of growing systems without global cache coherence and an increasingly diverse set of processing elements, we chose M^3 as the foundation of our work. Additionally, M^3 already supports byte-granular memory capabilities including their (selective) revocation (in contrast to CHERI).

The hardware architecture of M^3 is depicted in Figure 1. The key idea of M^3 is to introduce a new hardware component next to each processing element (PE), which is used as an abstraction for the heterogeneity of the PEs, ranging from general purpose cores to accelerators. This hardware component is called data transfer unit (DTU). All PEs are integrated into a network-on-chip (NoC) as prevalent in current multi- and manycore architectures [15, 39, 60]. The DTU represents the gateway for the PE to access remote memory (memories in other PEs or off-chip memory such as DRAM) and to exchange messages with other PEs. As such, the DTU enables a different isolation mechanism, called *NoC-level isolation*, that does not require the PEs to possess hardware features like MMUs and privileged mode execution to ensure isolation. Instead, since all communication between the PEs and all memory accesses are performed via the NoC, controlling the access to the NoC suffices to control and isolate the PEs.

The M^3 kernel runs on a dedicated PE, called *kernel PE*. The M^3 kernel is different from traditional kernels because it does not run user applications on the same PE based on user/kernel mode and entering the kernel via system call, interrupt, or exception. Instead, the kernel runs the applications on other PEs, called *user PEs*, and waits for system calls in the form of messages, sent by the applications via the DTU (red communication channel in Figure 1). Because there is only a single privileged kernel PE in M^3 this kernel PE quickly becomes the limiting factor when scaling to large systems.

Data Transfer Unit (DTU). The DTU provides a number of *endpoints* to connect with other DTUs or memory controllers. Endpoints can represent send, receive or memory endpoints. Establishing communication channels requires to configure endpoints to these representations. This can only be done by a privileged DTU. Initially all DTUs in the system are privileged and get downgraded by the kernel during boot up. Only the DTU of the kernel PE remains privileged. The kernel is required to establish the communication channels between applications (blue in Figure 1) which can be used by applications later on without involving the kernel.

Operating system. The M^3 OS follows a microkernel-based approach, harnessing the features of the DTU to enforce isolation at NoC-level. So far, it employs a single kernel PE to manage the system. M^3 implements drivers and OS services such as filesystems as applications, like other microkernel-based OSes. The execution container in M^3 is called *virtual PE* (VPE), which represents a single activity and is comparable to a single-threaded process. Each VPE has its own capability space and the M^3 kernel offers system calls to create, revoke, and exchange capabilities between the VPEs.

Services on M^3. Services are registered at the M^3 kernel and offer an IPC interface for clients. Additionally, clients can exchange capabilities with services. For example, M^3's in-memory file system, *m3fs*, offers an IPC interface to open files and perform meta operations such as `mkdir` and `unlink`. To access the files' data, the client requests memory capabilities from m3fs for specific parts of the file. The client can instruct the kernel to configure a memory endpoint for the memory capability to perform the actual data access via the DTU, without involving m3fs or the kernel again. This works much like memory mapped I/O, but with byte granular access. Reading the files' data via memory capabilities without involving the OS lends itself well for upcoming non-volatile memory (NVM) architectures.

3 Design

Our design strives to build a scalable distributed capability management for an operating system that uses multiple kernels. An application's scalability depends on two OS components: the kernel itself, especially the capability subsystem, and the OS services, e.g. a filesystem service. To investigate distributed capability management we concentrate on the kernel. The kernel sets up communication channels and memory mappings. How a service implementation uses the kernel mechanisms depends on the type of service. A copy-on-write filesystem for example can be implemented efficiently on top of a capability system with a sufficiently fast revoke operation.

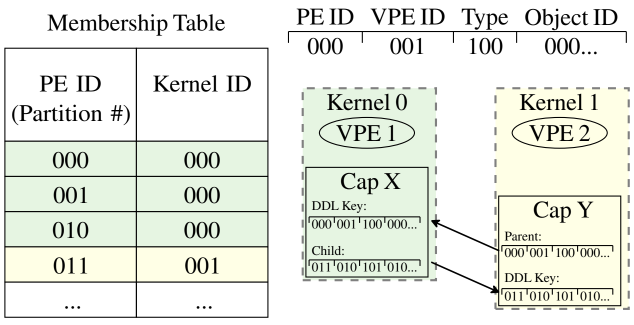

Figure 2: DDL addressing with globally valid DDL keys.

When an application performs a write it receives a mapping to its own copy of data and access to the original data has to be revoked. In a capability system with slow revocation it is questionable whether an efficient implementation of a copy-on-write filesystem is possible. The distributed capability system presented in this work shall lay a foundation for various service implementations, however, a discussion on the scaling of OS services is out of scope for this work.

3.1 System Overview

SEMPEROS employs multiple μ-kernels, each kernel running on a dedicated PE. This way we can distribute the handling of system calls to improve the scalability of the capability system. We use message passing as communication means between the kernels since we are not assuming a cache coherent system.

PE groups. To maintain a large number of PEs, we divide them into groups. Each group is managed by a single kernel; that means, every group has to contain at least one PE capable of executing a kernel. A simple general-purpose core is sufficient for this purpose. The group's kernel has exclusive control over all PEs of its group and manages the corresponding capabilities. The mapping of a PE's capabilities to a kernel is static in the current implementation of SEMPEROS because we do not yet support the migration of PEs. The unit of execution being scheduled on a PE is a virtual PE (VPE). A VPE is in general comparable to a process. All system calls of a VPE are handled by the kernel responsible for the PE, which the VPE is running on. If a system call does not affect any VPEs outside the group, only the group's kernel is involved in handling the request. Operations covering the VPEs of other groups involve their kernels as well. A more detailed view on the communication to handle system calls is given in Section 3.3.

Distributed state. The system state consists of the hardware components (available PEs), the PE groups, and the resource allocations and permissions, represented as capabilities. The capabilities represent VPEs, byte-granular memory mappings, or communication channels. Our high-level approach to manage the aforementioned kernel data is to store it where the data emerges and avoid replication as far as possible. Thereby, we minimize the shared state, which reduces the communication required to maintain the coherence.

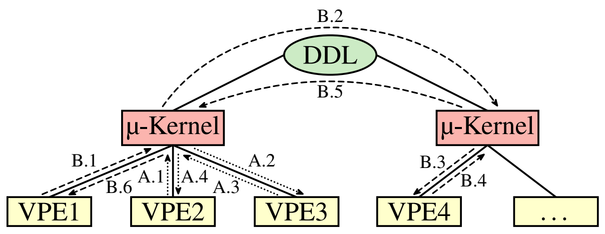

Figure 3: Two VPEs establish a communication channel. Sequence A shows the group-internal communication, whereas sequence B is group-spanning involving two kernels.

3.2 Distributed Data Lookup (DDL)

The distributed data lookup is our capability addressing scheme and the mechanism to determine the location of kernel data. Each kernel object or capability which needs to be referable by other kernels is given a *DDL key* which acts as its global ID. In the top right corner of Figure 2 we illustrate how we split the key's value into several regions representing the following: the *PE ID* and *VPE ID*, denoting the creator of the object, and the *Type* and *Object ID*, describing the object itself. To clarify the concept the amount of digits is deliberately kept smaller than our system requires in practice. We use the PE ID to split the key space into multiple partitions. Each PE in the system is allocated to one such partition, which in turn are assigned to kernels individually. The mapping of partitions to the kernels designates the PE groups and is stored in a membership table which is present at each kernel and depicted on the left of Figure 2. To support the migration of PEs, in SEMPEROS, the mappings in the membership table would have to be updated at all kernels of the system However, our current implementation does not support migration yet.

With this addressing scheme we are able to reference objects, for example capabilities, across the whole system which is a key enabler for our capability scheme. The lower right part of Figure 2 illustrates how the relations between the capabilities are tracked by this means. It depicts a situation in which two applications, VPE 1 and VPE 2, are residing in different PE groups, thus managed by different kernels. The (simplified) excerpt of the kernels' internal capability mappings shows that VPE 1 has a capability Cap X and delegated it to VPE 2, which in turn created Cap Y. This group-spanning relation is tracked using the DDL keys.

3.3 System Call Handling

System calls are implemented by messages sent to the kernel PE. Some of the actions taken by the kernel receiving a system call involve agreement between the kernels depending on the involved VPEs. Hence, we divide the handling of system calls into group-internal and group-spanning operations. The different message sequences are outlined in Figure 3.

Group-internal operations. Sequence A in Figure 3 depicts the establishing of a communication channel between two

VPEs (2 & 3) in the same group. Such an operation only involves resources managed by a single kernel, thus it is called a group-internal operation. The sequence starts with a message to the kernel (A.1), which is the counterpart to the traditional mode switch. The connection request is forwarded to VPE3 (A.2) which then responds to the kernel. Depending on the response, the kernel hands out the appropriate capabilities and informs VPE2 (A.4). Once the endpoint for the exchanged communication capability is configured, the kernel is not involved in further communication.

Group-spanning operations. Sequence B in Figure 3 shows the message flow when VPEs of different PE groups establish a communication channel. The second kernel is involved in the operation because it is in charge of VPE4's capabilities. This is where our distributed capability protocol will be used. After receiving the system call in step B.1, the first kernel uses the DDL to determine which kernel is responsible for VPE4 in step B.2 and forwards the request. Steps B.3 and B.4 are identical to A.2 and A.3 of the group-internal operation. After these steps, the channel at VPE4's side is prepared which the first kernel indicates to VPE 1. As for the group-internal operations, the communication via the created channel does not involve the kernel anymore after the endpoint has been configured.

3.4 Capabilities

SEMPEROS employs capabilities for managing permissions. Each VPE has its own set of capabilities, describing the resources it can access. To share resources with other VPEs, capabilities can be exchanged. In SEMPEROS delegating a capability starts with a system call by the supplying VPE indicating to the kernel which capability should be delegated. The kernel makes a copy of the selected capability and adds that copy to the capability's children. The copied capability (the child) is handed over to the VPE which shall receive the new access rights. If the sharing is no longer desired, the access rights to the resource can be withdrawn recursively by revoking the capability. These operations require action of the kernel.

From the kernel's perspective, a capability references a kernel object, a VPE, and other capabilities. The kernel object is the resource this capability permits access to and the reference to the VPE (which is a kernel object of its own) tells the kernel who holds these access rights. Individual references to other capabilities are maintained to track sharing as is done by the mapping database in other microkernel-based systems [30, 36, 40, 62]. SEMPEROS keeps sharing information in a tree structure which is used to enable recursive revocation. In the capability tree, capabilities of different VPEs are interlinked, as indicated in Figure 2.

Challenges in a distributed setting. Capabilities are modified by multiple operations, requested via system calls from applications. For example, when creating a VPE, a capability to control the new VPE is delegated to the parent VPE. We call such actions *capability modifying operations* (CMO). Running a system with multiple independent kernel instances introduces new properties of capability handling:

1. Multiple CMOs can be in flight at the same time.
2. CMOs can involve the modification of capabilities owned by other kernel instances.
3. A capability managed by one kernel can reference a kernel object owned by another kernel.

The first property requires to assure that the modifications of one kernel do not overlap with changes of another kernel. The second and third property are results of our system's distributed nature. Resources such as a service, which is resembled by a service capability, could be used by VPEs of different PE groups. Consider the connection establishment to a service. Assuming that the service is controlled by kernel 1 and the connecting VPE by kernel 2, kernel 2 would create a session capability. A session can only be created between a client and a service; hence, the client's session capability is listed as a child of the service capability. This modification of the service capability's list of children involves the other kernel, because the service capability is owned by kernel 1.

Since capabilities are used to control access to resources, we host a capability at the kernel which owns the resource. Yet, this attribution is not always obvious. The example of a session capability illustrates the third property. One could argue, that the session is a resource which is used by both, the service and the client; thus, any of the two corresponding kernels could actually be responsible for the session. To avoid the overhead of coordination between multiple resource owners, we allow only one kernel to be the owner of a resource.

4 Implementation

SEMPEROS implements a distributed capability scheme with multiple kernels in order to scale to large numbers of PEs. We based SEMPEROS on M^3 [5]. SEMPEROS adds PE groups to the base system, requiring coordination of the kernels which we implement by so called inter-kernel calls explained in the following Section 4.1. Furthermore, SEMPEROS is implemented as a multithreaded kernel for reasons explained in Section 4.2. The transparent integration of the PE groups into a single system presented to the applications requires the kernels to implement a distributed capability system described in Section 4.3.

4.1 Inter-Kernel Calls

The system call interface of SEMPEROS did not change compared to M^3 though the action to be taken by the kernel changed to incorporate the coordination with other kernels. The kernels in SEMPEROS communicate via messages adhering to a messaging protocol. We call this type of remote procedure calls *inter-kernel calls*. These calls can be split into three functional groups: (1) messages to start up and shutdown kernels and OS services, (2) messages to create connections to the services in other PE groups, and lastly, (3) messages used to exchange and revoke capabilities across PE-group boundaries. Messages of the last two groups are part of the distributed capability protocol.

The DTU, which is used to send and receive messages, provides only a limited number of message slots. If this limit is exceeded then the messages will be lost. To prevent this, we limit the number of in-flight messages between two kernels. We dedicate a certain number of DTU endpoints for the kernel-to-kernel communication, which also determines the maximum number of kernels supported by the system. We keep track of free message slots at each kernel to avoid the message loss.

4.2 Multithreaded Kernel

The kernel needs to split some operations, e.g. revocation or service requests across the PE groups, into multiple parts to prevent deadlocks. For instance, a revocation might run into a deadlock in the following situation: three capabilities are involved forming the capability tree: $A_1 \rightarrow B_2 \rightarrow C_1$. The index indicates the kernel which owns the capability. If A_1 is revoked, kernel 1 contacts kernel 2 which in turn contacts kernel 1 again to revoke capability C_1. If kernel 1 would block on the inter-kernel call to kernel 2, the system would end up in a deadlock because kernel 2 is waiting for kernel 1 to respond. While this can be implemented as an event-driven system, this involves the danger to loose the overview of the logical flow of complicated operations like the revocation.

Therefore, we decided to use cooperative multithreading within the kernel. This approach allowed us to implement such capability operations sequentially with dedicated preemption points in between, which made it comparatively easy to reason about the code. Note that, in contrast to simultaneous multithreading, SEMPEROS only executes one thread per kernel at a time because it executes on one single-threaded core. The preemption points do not only prevent deadlocks, but also allow to process other system calls or requests from other kernels until the suspended operation can be continued.

To prevent the denial-of-service attacks on the kernel, the kernel cannot spawn new threads on behalf of system calls. Instead, a fixed number of threads needs to suffice. We create a kernel's thread pool at start up. The size of the pool is determined by the number of system calls and kernel requests which can arrive at the same time. It is calculated as:

$$V_{group} + K_{max} * M_{inflight} \qquad (1)$$

Since each VPE can issue only one system call at a time, the kernel needs one thread per VPE in its PE group, denoted as V_{group}. The number of kernel requests is limited by the maximum amount of kernels in the system, denoted as K_{max}, multiplied by the maximum number of in-flight messages between two kernels, denoted as $M_{inflight}$.

4.3 Distributed Capability Management

SEMPEROS uses capabilities to control the access to resources such as VPEs, service connections, send/receive endpoints and memory. Applications can *create*, *use*, *exchange*, and *revoke* capabilities. Creation means to create a new capability for a given resource (e.g., memory) and usage means to use a previously created capability without changing it (e.g., configure a DTU endpoint for a send capability). Exchanges and revokes are *capability modifying operations* (CMO), requiring the most attention. Exchanging capabilities allows two VPEs to share resources and it comes in two flavors: a capability can be *delegated* to another VPE, and *obtained* from another VPE. Capability exchanges can be undone with the revoke operation. Revocation is performed recursively, that is, if VPE V_1 has delegated a capability to VPE V_2, which in turn has delegated it to V_3 and V_4, and V_1 revokes its capability, it is revoked from all four VPEs.

As in other capability systems [30, 36, 40, 62], the recursive revoke requires a way to track former exchanges. The kernel uses a so-called *mapping database* for this purpose. In SEMPEROS each capability has a parent and a list of children to explicitly track all such links. These tree relations can span multiple kernels; hence, we use DDL keys to identify and locate the capabilities across all kernels. The mapping database is updated on every CMO. In a multikernel setting multiple CMOs can be started concurrently potentially involving the same capability. We next describe how inconsistent updates on the mapping database are prevented if multiple CMOs run in parallel.

4.3.1 Interference between CMOs

Exchanging a capability consists of two actions: (1) creating a new capability based on the donor's capability and (2) inserting the new capability into the capability tree. The latter requires to store a reference to the parent capability and to update the parent's list of children. Revoking a capability requires to revoke all of its children and to remove it from the parent's list of children. Both need to perform inter-kernel calls in case capabilities reside at other kernels, possibly leading to interference.

An important precondition for all operations is that messages between two kernels need to sustain ordering. More specifically, if kernel K_1 first sends a message M_1 to kernel K_2, followed by a message M_2 to kernel K_2, then K_2 has to receive M_1 before M_2.

Table 2 shows an overview of all combinations and their effects. The operation in the leftmost column is started first and overlaps with the operation in the topmost row. The following walks through the combinations and describes the effects.

Serialized. Overlapping exchange operations do not present a problem for our scheme because they serialize at one kernel. For example, if two VPEs obtain a capability from VPE V_1, these operations serialize at the kernel that manages V_1. In general, each VPE can only perform one system call at a time preventing two parallel delegates initiated by the same VPE.

1st \ 2nd	Obtain	Delegate	Revoke/Crash
Obtain	✓ Serialized	✓ Serialized	! Orphaned
Delegate	✓ Serialized	✓ Serialized	↯ Invalid
Revoke	! Pointless	! Pointless	↯ Incomplete

Table 2: Types of interference with overlapping CMOs.

However, during an exchange operation initiated by a VPE V_1 other VPEs could exchange capabilities with V_1, which again serializes at the kernel that manages V_1.

Orphaned. The obtain operation needs to ask the capability owner for permission before being able to obtain the capability. If the owner resides at a different kernel this requires an inter-kernel call. Before its completion nothing is changed in the obtainer's capability tree. However, the obtainer could be killed while waiting for the inter-kernel call. This leaves an orphaned child capability in the owner's capability tree, in case the owner agreed to the exchange. The orphaned capability cannot be accessed by anyone, but it wastes a bit of memory, which will be freed the latest when its parent capability is revoked.

Invalid. The delegate operation is similar to obtain regarding leaving the delegator's capability tree untouched until the inter-kernel call returns successfully. However, if the delegator is killed while waiting for the inter-kernel call, the receiving VPE might have already received the capability. This constitutes a problem, because the child of the capability in delegation does not yet exist in the delegator's capability tree. That is, although all capabilities of the delegator are revoked, the delegated capability stays valid at the receiving VPE.

Incomplete. The naive implementation of the revoke operation would simply perform a depth-first walk through the capability tree, remove local capabilities on its own and perform the inter-kernel calls to remove remote capabilities. However, if two revoke operations run in parallel on overlapping capability subtrees, this approach results in early replies to revoke system calls, that is, acknowledgements of incomplete revokes.

For instance, let us consider the following capability tree with the owning kernel as the index: $A_1 \rightarrow B_2 \rightarrow C_3$. If a VPE requests the revoke of A_1, kernel K_1 performs a call to K_2 to revoke the remaining part of the tree. If another VPE requested the revoke of B_2 in the meantime, K_2 does not know B_2 anymore, potentially leading to an early response to K_1. The reason is that K_2 might still be waiting for K_3 to revoke C_3. Since applications have to rely on the semantic that completed revokes are indeed completed, we consider this behavior unacceptable.

Pointless. The revoke operation requires inter-kernel calls if the capability tree spans multiple kernels. Hence, VPEs might request capability exchanges of not yet revoked capabilities within this tree. This does not lead to inconsistencies because these capabilities would be revoked as soon as the running revoke operation continues. However, the exchange is pointless because it is already known that the capabilities will be revoked afterwards.

4.3.2 Capability Exchange

This section details the capability exchange operations to address the problems described in the previous section. As already mentioned, the beginning of obtain and delegate is similar. If a VPE (V_1) requests an exchange, the corresponding kernel (K_1) checks whether the other party (V_2) is in the same PE group. If so, the operation is handled by K_1. If V_1 and V_2 are in different PE groups, K_1 forwards the exchange request to the second kernel (K_2). K_2 asks V_2 whether it accepts the capability exchange. If V_2 denies the exchange, the operation is aborted and a corresponding reply is sent to K_1. Otherwise, we distinguish between obtain and delegate:

(1) Obtain:. V_2's capability (C_2) will become the parent of V_1's new capability (C_1). Hence, C_1 will be added to C_2's list of child capabilities. Afterwards, K_2 sends a reply to K_1. As outlined previously, if V_1 was killed in the meantime, C_2 stays in the child capability list as an orphaned capability. To prevent a permanent memory waste, we let K_1 send a notification to K_2 on behalf of K_2's reply for the obtain operation in case V_1 was killed.

(2) Delegate:. K_2 creates a new capability (C_2) for V_2 with C_1 as its parent. If C_1 was revoked in the meantime, V_2's resource access through C_2 would be unjustified. To avoid this, we implement delegation with a two-way handshake. Instead of inserting C_2 into V_2's capability tree, K_2 only sends a reply to K_1. After that K_1 adds C_2 to C_1's list of children and sends an acknowledgement back to K_2 to actually insert C_2 into V_2's capability tree.

Note that the two-way handshake creates an orphaned capability if V_2 is killed while waiting for the acknowledgement of K_1. As for obtain, we handle this case by sending an error back to K_1 to allow a quick removal of the orphaned capability.

4.3.3 Capability Revocation

Like the capability exchange, the revocation requires inter-kernel calls in case the capability tree spans multiple kernels. To keep the kernel responsive it should not wait synchronously for the reply of another kernel. Instead, the revoke should be paused using the threading infrastructure introduced in Section 4.2. However, in contrast to the exchange operation, the number of inter-kernel calls for a revoke can be influenced by applications. For example, two malicious applications residing in different PE groups could exchange a capability back and forth, building a deep hierarchy of capabilities at alternating kernels. Revoking this capability hierarchy would lead to inter-kernel calls sent back and forth between the two kernels. Thus, the naive approach of spawning a new thread for every incoming revoke inter-kernel call cannot be used, because this would enable denial-of-service attacks. Our solution uses a maximum of two threads per kernel to avoid this attack.

To avoid acknowledgements of incomplete revokes, our algorithm uses two phases, similar to mark-and-sweep [46]. Algorithm 1 presents a high-level overview of the approach. The function `revoke_syscall_hdlr` is executed by the kernel that receives the revoke system call. First, it calls `revoke_children`, which recursively marks all local capabilities and sends inter-kernel calls for remote capabilities. Each capability maintains a counter for outstanding kernel replies. If it encountered any remote capabilities, the function `wait_for_remote_children` waits for the kernel replies by pausing the thread.

The inter-kernel call is handled by `receive_revoke_request`, which will also call `revoke_children`. In this case, the thread will not be paused to stay at a fixed number of threads. Instead, the thread calls `receive_revoke_reply` in case there are no outstanding kernel replies and returns. The function `receive_revoke_reply` is also called whenever a reply to an inter-kernel call is received, which first updates the counter of the capability accordingly. If there are no further outstanding kernel replies, it deletes the capability tree starting at the given capability. Afterwards, it wakes up the syscall thread or sends a reply, depending on whether this kernel started the revoke operation or participated due to an inter-kernel call.

To keep the pseudo code brief it does not show how already running revocations for a capability are handled. In this case, `revoke_syscall_hdlr` will also wait for the already outstanding kernel replies to prevent acknowledgement of an incomplete revoke. Furthermore, the two phases allow us to immediately deny exchanges of capabilities that are in revocation, which prevents pointless capability exchanges.

Algorithm 1: Capability revocation

```
1  Function revoke_syscall_hdlr(capability)
2      revoke_children(capability)
3      wait_for_remote_children()
4  Function revoke_children(capability)
5      mark_for_revocation(capability)
6      foreach child of capability do
7          if child is local then
8              revoke_children(child)
9          else
10             send_revoke_request(child)
11         end
12     end
13 Function receive_revoke_request(capability)
14     revoke_children(capability)
15     receive_revoke_reply(capability) // see line 10
16 Function receive_revoke_reply(capability)
17     if all revoke requests are serviced then
18         delete_tree(capability)
19         if initiator then
20             notify_syscall_hdlr_thread()
21         else
22             send_revoke_reply()
23         end
24     end
```

Operation	**Scope**	**SemperOS** (cycles)	**M^3** (cycles)	**Increase**
Exchange	Local	3597	3250	10.7%
Exchange	Spanning	6484	—	—
Revoke	Local	1997	1423	40.3%
Revoke	Spanning	3876	—	—

Table 3: Runtimes of capability operations.

5 Evaluation

5.1 Experimental Testbed

We evaluate SemperOS using the gem5 system simulator [14], which enables us to evaluate the hardware/software co-designed capability system and perform experiments on systems larger than currently available. The system is composed of 640 out-of-order x86_64 cores, which are clocked at 2 GHz. However, the cores used for applications could also be exchanged with any other architecture or accelerator. Each core is equipped with a DTU similar to the one used in Asmussen et al.'s work [5]. We modified the mechanism to store incoming messages to support delayed replying which enables us to interrupt kernel threads. Messages are kept in a fixed number of slots. Each DTU provides 16 endpoints with 32 message slots each. Kernel PEs use one endpoint to send messages to other kernels, one endpoint to send messages to services, and 14 endpoints to receive messages. Six of the receiving endpoints are used for the system calls. Each kernel can handle up to 192 PEs in the current implementation since each VPE can only issue one (blocking) system call at a time. Eight endpoints are used to receive messages from other kernels. In contrast to system calls, the inter-kernel calls are non-blocking and we limit the number of in-flight messages to four messages per kernel. Thus, at most 64 kernel PEs are supported.

5.2 Microbenchmarks

Capability exchange and revocation. To examine the exchange and revocation of capabilities we start two applications where the second application obtains a capability from the first, followed by a revoke by the first application. We distinguish two scopes for these operations: group-local and group-spanning. In the group-local case one kernel manages both applications and their capabilities. The group-spanning case involves two kernels, each handling one application.

Table 3 lists the execution times in cycles for exchanging and revoking capabilities in the group-local and group-spanning case. We can only compare the group-local case to M^3, because in M^3 there is only one kernel. To support multiple kernels, SemperOS references parent and child capabilities via DDL keys instead of plain pointers. Analyzing the DDL key to determine the capability's owning kernel and VPE introduces overhead in the local case. Group-spanning operations involve another kernel, which almost doubles the time of exchanges and revokes. This suggests, that applications should be assigned to PE groups such that the group-spanning operations are minimized.

Chain revocation. In the chain revocation benchmark we measure the time to revoke a number of capabilities forming a chain. Such a chain emerges when a capability is exchanged with an application which in turn exchanges this capability

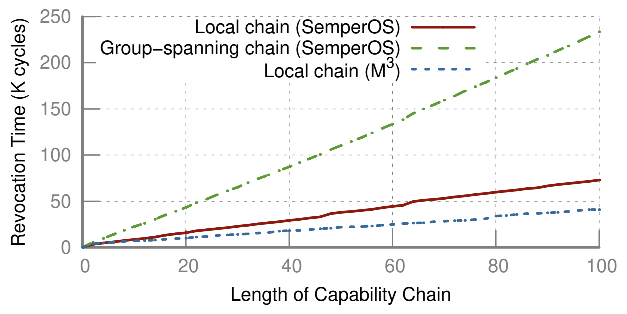

Figure 4: Revoking capability chains of varying sizes.

again with another application and so on. Figure 4 depicts the time to revoke such capability chains depending on their length. A local chain comprises only applications managed by one kernel and can again be compared to M^3. As the previous microbenchmarks showed, revocation in SEMPEROS needs about twice the time compared to M^3 due to the added indirections.

The group-spanning chain depicts a scenario in which an ill-behaving application repeatedly exchanges a capability between two VPEs, which are managed by different kernels. This creates a circular dependency between the two involved kernels during revocation. However, as described in Section 4.3.3, this is not a problem for our revocation algorithm. In particular, only the kernel thread for the revoke system call is blocked during the operation. Still, the revocation of a group-spanning chain takes about three times longer than revoking a group-local chain, because messages are sent back and forth between the two kernels.

Tree revocation. This microbenchmark resembles a situation, in which an application exchanges a capability with many other applications, for example, to establish shared memory. This results in a capability tree of one root capability with several children. Figure 5 shows the performance of the revocation depending on the capability count and their distribution among kernels. The line labeled with 1 + 0 Kernels represents the local scenario in which the whole capability tree is managed by one kernel. For all other lines, the second number indicates the number of kernels the child capabilities have been distributed to. After exchanging the capabilities, the application owning the root capability revokes the capability tree. Figure 5 illustrates that the revocation scales to many capabilities and kernels. It also shows that our current implementation can take advantage of multiple kernels by performing the revoke in parallel, but the effect is rather small. It currently leads to a break-even at 80 child capabilities, when comparing the local revocation time with a parallel revocation with 12 kernels. However, we believe that this can be further improved by the use of message batching. So far, the kernel managing the root capability sends out one message for each child capability.

5.3 Application-level Benchmarks

We next perform application-level benchmarks to examine the scalability of SEMPEROS in more realistic settings.

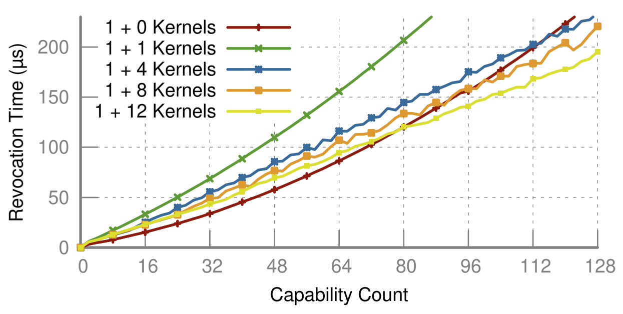

Figure 5: Parallel revocation of capability trees with different breadths utilizing multiple kernels.

5.3.1 Experimental Setup

Applications. We use seven different applications to analyze the scalability: *tar* and *untar* pack or unpack an archive of 4 MiB containing five files of sizes between 128 and 2048 KiB. The *find* benchmark scans a directory tree with 80 entries for a non-existent file. The *SQLite* database and the *LevelDB* key-value store both create a table to insert 8 entries into it and select them afterwards. The PostMark mailserver application resembles a heavily loaded mail server, thus does a lot of operations on the mail files. (In addition, we evaluated *Nginx* Webserver in Section 5.3.3.) Note that we were forced to use rather short running applications to keep the simulation times of gem5 acceptable (e.g. SQLite required five days on a 48-core machine). The selected applications are well suited for this evaluation since they make heavy use of the OS in various ways. In particular they use the in-memory filesystem service which implements file access by handing out the memory capabilities to the clients so they can access the memory region in which the requested file is stored. More specifically, the filesystem service hands out a memory capability to a range of the file's contents. If the application exceeds this range, for example by appending to the file, it is provided with an additional memory capability to the next range. When the file is closed again, the memory capabilities are revoked.

Table 4 lists the number of capability operations for the individual benchmark applications. We show the numbers for a single benchmark instance and 512 parallel instances. The capability operations per second for 512 benchmark instances are retrieved when employing 64 kernels and 64 filesystem services; we will explain what this means in the following paragraph on our methodology. The tar and untar benchmarks are memory-bound applications exposing a regular read and write pattern which requires the filesystem service to hand out several memory capabilities throughout the benchmark execution. The find benchmark mainly stresses the filesystem service by doing many stat calls to examine the directory's metadata. The small database engine of SQLite exhibits a more compute intensive behavior with several bursts of capability operations when opening and closing the database and the database journal whereas the LevelDB key-value store accesses its data files with a higher frequency resulting

Benchmark	Cap. ops	Cap. ops/s	Cap. ops	Cap. ops/s
# of instances	1		512	
tar	21	7,295	10,752	191,703
untar	11	4,012	5,632	100,772
find	3	1,310	1,536	27,096
SQLite	24	5,987	12,288	207,072
LevelDB	22	8,749	11,264	201,204
PostMark	38	21,166	19,456	348,285

Table 4: Number of capability operations for the selected applications. Values shown for 1 and 512 parallel benchmark instances. The capability operations per second are the average rate of capability operations over the runtime.

in more capability operations per second. PostMark does little computation and operates on many files resulting in the highest load for the capability system.

Performance metric. We use the system call tracing infrastructure introduced by Asmussen et al. [5] to run the benchmarks. We run an application on Linux, trace the system calls including timing information, and replay the trace on SEMPEROS while checking for correct execution. We account for the system calls which are not supported by our system yet by waiting for the time it took to execute them on Linux. However, all relevant system calls (especially those to interact with the file system) are executed. We replay the same trace multiple times in parallel, which is denoted as number of benchmark or application instances in the graphs. We assess scalability using the parallel efficiency of these benchmark instances. In a perfectly scaling system, a benchmark instance will have the same execution time when running alone as when running with other instances in parallel. However, due to resource contention for the kernel and for hardware resources like the interconnect and the memory controller, each instance will need more time if multiple of them are executed in parallel. The discrepancy in runtime is shown by *parallel efficiency*.

Methodology. There are two main factors influencing the scalability of applications running in a μ-kernel-based system: the OS services and the kernel. The OS service used by the examined applications is the m3fs filesystem. To concentrate our analysis on the scalability of the kernel, especially the distributed capability management, we simplify scaling of the m3fs service by adding more service instances, each having its own copy of the filesystem image in memory. We exclude accessing the actual memory locations of the files because our current simulator does not include a scalable memory architecture yet. Instead we let the application compute for the amount of time the access would have taken, assuming a non-contended memory controller. We argue that this still produces useful results since we do not want to show the scalability of the memory architecture but of the distributed capability scheme. Furthermore, a non-contended memory puts even more burden on the OS because capability operations might occur with higher frequency.

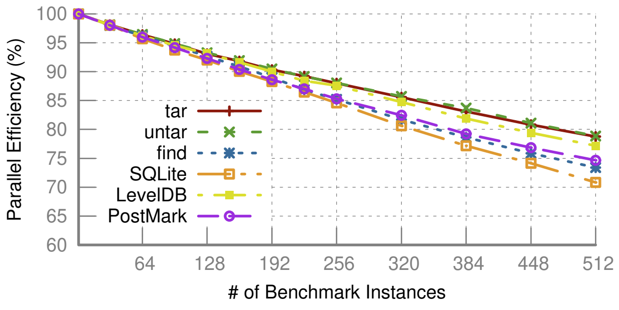

Figure 6: Parallel efficiency of all six applications using 32 kernels and 32 file service instances.

5.3.2 Results

Scalability. Figure 6 depicts the parallel efficiency of the six applications when distributing them equally between 32 kernels and 32 filesystem services. With this configuration the tar benchmark already reaches an efficiency of 78% when running 512 instances in parallel. However, SQLite achieves only 70%, which is not the optimal configuration for this type of application as we will show in the next measurement (see Figure 7). We next discuss how to determine a fitting configuration for an application.

Service dependence. To determine the number of services required to scale an application we set the number of kernels to a high number and then gradually increase the number of services. As long as there are less services than kernels, services are shared between PE groups. Kernels which host a service in their PE group prefer to connect their applications to the service in their PE group over a service in another PE group. Figure 7 shows the parallel efficiency for tar and SQLite depending on the number of services.

The tar benchmark is not very dependent on the filesystem service, which can be inferred from the fact that using 48 services does not pose any improvement over 32 services. In fact, it seems already fair enough to use only 16 service instances. SQLite shows a higher dependence on the number of services. For example, increasing the number of service instances from 16 to 32 leads to further improvement of 9 percent points.

Kernel dependence. Similarly to the dependence on the number of services, we now show the influence of the number of kernels. Figure 8 depicts the parallel efficiency of PostMark and LevelDB using a fixed number of services. LevelDB exhibits smaller improvements when employing more than 16 kernels compared to PostMark, indicating that PostMark is even more susceptible to the number of kernels. However, all applications show a relatively high sensitivity to the number of kernels, which in fact are mostly handling capability operations. This confirms our expectation that a scalable distributed capability system is a vital part of a fast μ-kernel-based OS for the future hardware architectures. The analysis so far only involved tuning for parallel efficiency, which is analogous to optimize for execution time. We next discuss the efficient usage of PEs.

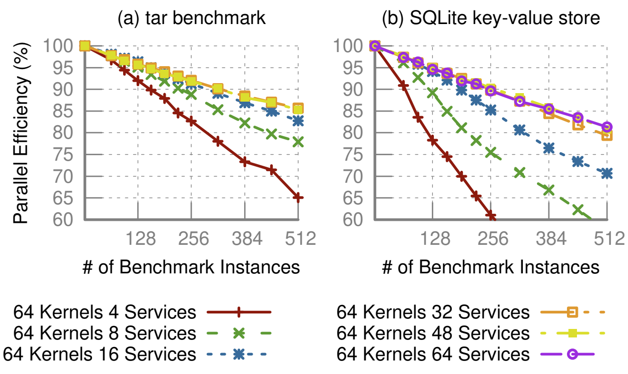

Figure 7: Service dependence: Parallel efficiency of tar and SQLite with fixed number of kernels.

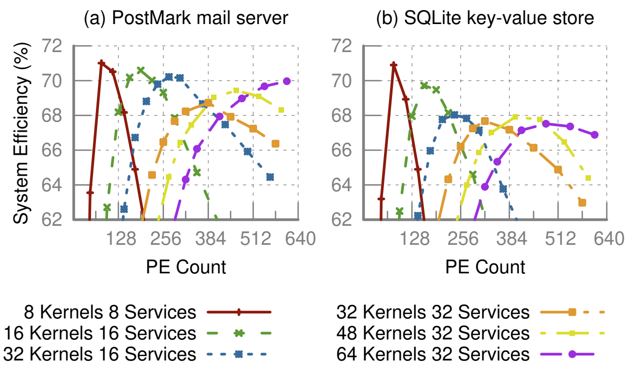

Figure 9: System efficiency of PostMark and SQLite with different configurations.

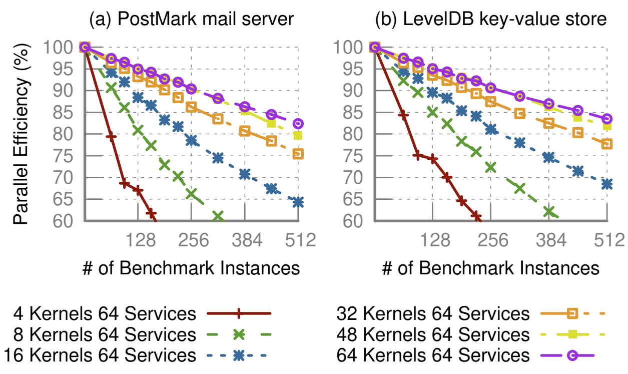

Figure 8: Kernel dependence: Parallel efficiency of PostMark and LevelDB with fixed number of services.

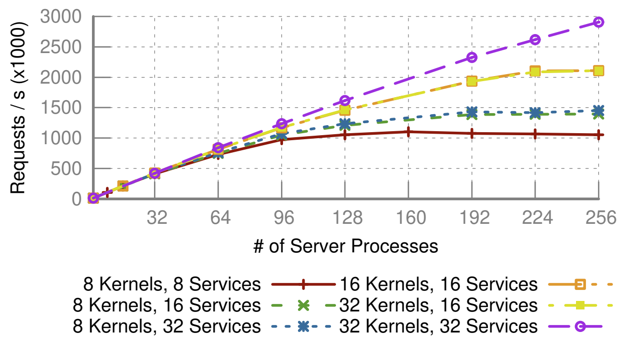

Figure 10: Scalability of the Nginx webserver.

System efficiency. If we consider the whole system and account for the PEs used by the OS with an efficiency of zero, the optimal configurations change. We call this measure the *system efficiency*, which is depicted in Figure 9. Instead of showing the efficiency only in relation to the benchmark instances executed we relate them to the total number of PEs. By means of this metric we can tune a system for throughput and determine the optimal number of kernels and services for an application depending on the number of PEs available. For SQLite this implies to choose 16 kernels and 16 service instances if the system had 192 PEs, but if the system would consist of 256 PEs we would run it with 32 kernels and 16 services.

5.3.3 Server Benchmark

We next detail the results for the Nginx webserver [53]. We used our system call tracing infrastructure to record the behavior of Nginx on Linux when handling requests. We stressed Nginx similar to the Apache ab benchmark [1] by introducing PEs that resemble a network interface. These PEs constantly send out requests to our webserver processes running on separate PEs. These PEs replay the trace upon receiving a request and send the response back. Figure 10 depicts the number of requests per second of all webserver PEs. Despite this OS-intensive benchmark, the number of requests scales almost linearly when employing 32 kernels and 32 services. Using less resources for the OS flattens the graph.

6 Related Work

Capability systems. The evaluation of previous capability systems typically resorted to performance measurements of single core or small systems. Since many capability systems, such as Mach, Fluke or EROS [19,23,56] are based on μ-kernels which had to prove their enhanced efficiency over previous μ-kernel generations [29,41], their kernel mechanisms like inter-process communication, system-call performance, context-switch overhead or process-creation time have been measured to demonstrate their in fact competitive performance. With these measurements it is only partly possible to derive the performance of the capability subsystem. So far the only work which included capabilities in a distributed setting is Barrelfish [11] but only parts of the capability subsystem's scalability can be concluded from the reported results. Barrelfish's two-phase commit approach to reach agreement (determining the relations between capabilities) requires broadcasting to every other kernel in the system, which is different from our approach. The revocation in Barrelfish uses a mark-sweep algorithm and so called *delete cascades* which also require to broadcast to every kernel if capabilities have cross-kernel relations because these are not stored explicitly in Barrelfish [25]. Even though this broadcast operation can be tuned to fit the interconnect of the machine it is running on [33], it is unknown how well it performs in conjunction with their capability scheme.

Other capability systems like Capsicum [65] and CHERI [67] emphasize their sandboxing features and compatibility to existing software by executing application benchmarks. However, these do not include any assessment of large scalable systems. Further, CHERI does not support revocation, thus eliminating the overhead of tracking capability relations.

Operating systems. Apart from a scalable capability subsystem the OS also has to entail mechanisms to drive large possibly heterogeneous systems. The monolithic architecture of Linux, which runs a shared-memory kernel on homogeneous cores, has many scalability bottlenecks [16, 17, 26]. Developers try to counteract that by utilizing scalable data structures like RCU [47] within the kernel. To investigate more profound changes researchers built frameworks like K42 [37] to enable development and testing of new approaches like clustered objects [3]. Song et al. proposed Cerberus [61] which runs multiple Linux instances on top of a virtualization layer.

Systems like Popcorn Linux [8–10], and K2 [42] adapt Linux for heterogeneous ISAs. These systems also run multiple Linux instances closely resembling a distributed system. Rack-scale operating systems like LegoOS employ multiple distributed components to manage disaggregated resources [55]. They can benefit from our approach when combined with capabilities.

Barrelfish proposed the multikernel approach which aims to improve the scalability and support for many heterogeneous cores by constructing the OS as a distributed system. Cosh [12], a derivative of Barrelfish, demonstrated how to share and provide the OS services across different domains. While Cosh defines an interface how to communicate between different coherence islands and adds guarantees regarding memory accesses after sharing, it is not discussing the underlying capability system. Barrelfish/DC [68] examined the separation of kernel state from a kernel instantiation to provide an elastic system. Fos [66] targets manycore systems by proposing the concept of service fleets to provide OS services via spatially distributed servers. Importantly, these OSes are based on communication over message passing and do not assume cache coherent shared memory. Our system SemperOS shares the same two design principles: (1) the multikernel approach, and (2) communication via message passing. However, the design of Barrelfish and fos require executing a kernel on every core. Whereas, we based our work on M^3 [5] which runs the kernel only on a single dedicated core. This allows us to explore another design point in the multikernel design space in which the capabilities of several processing units are managed by one kernel which has to coordinate with other kernels to scale to large systems.

The philosophy of providing OS services without executing a kernel on every core has also been explored in NIX [7], which is based on Plan 9 [52]. NIX proposes an OS with a concept of application cores which do not execute a kernel to prevent OS noise. However, the communication in NIX is based on shared memory. Similarly, Helios [50], an extension of Singularity [22], minimizes the kernel requirement for some cores to a smaller satellite kernel.

Motivated by the recent trends in hardware, in a similar spirit but with a different focus, new OSes such as Arrakis [51], IX [13], and Omnix [57] have been proposed. These OSes share a similar design philosophy to SemperOS where we aim to provide applications direct control of the underlying hardware to improve the performance.

Alternatively, there are several proposals to support OS services for one specific type of accelerator. For instance, GPUfs [58], GPUNet [35], and PTask [54] are designed for GPUs. Likewise, BORPH [59], FPGAFS [38], etc. are designed to support FPGAs. In contrast, using M^3 as our foundation allows us to support different types of accelerators and general purpose heterogeneous cores as first-class citizens.

7 Conclusion

In this paper, we presented a HW/SW co-designed distributed capability system based on M^3. More specifically, we presented a detailed analysis of distributed capability management, covering the inconsistencies which can arise in a distributed multikernel setting where concurrent updates to capabilities are possible. Leveraging the results of this investigation we devised efficient algorithms to modify capabilities in a scalable and parallel manner.

We implemented these algorithms in our microkernel-based OS, SemperOS, which employs multiple kernels to distribute the workload of managing the system. We evaluated the distributed capability management protocols by co-designing the HW/SW capability system in the gem5 simulator [14]. Our evaluation shows that there is no inherent scalability limitation for capability systems for running real applications: Nginx, SQLite, PostMark, and LevelDB. In particular, we showed that SemperOS achieves a parallel efficiency of 70% to 78% when running 512 applications and dedicating 11% of the system's cores to the OS.

Software availability. SemperOS will be open-sourced at https://github.com/TUD-OS/SemperOS.

8 Acknowledgements

We would like to thank our shepherd, Gernot Heiser, and the anonymous reviewers for their helpful suggestions. This work was funded through the German Research Council DFG through the Cluster of Excellence Center for Advancing Electronics Dresden (cfaed), and by the German priority program 1648 "Software for Exascale Computing" via the research project FFMK, and by public funding of the state of Saxony/Germany.

References

[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. Accessed: May, 2018.

[2] ANDERSON, M., POSE, R., AND WALLACE, C. S. A password-capability system. *The Computer Journal* (1986).

[3] APPAVOO, J., SILVA, D. D., KRIEGER, O., AUSLANDER, M., OSTROWSKI, M., ROSENBURG, B., WATERLAND, A., WISNIEWSKI, R. W., XENIDIS, J., STUMM, M., AND SOARES, L. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems (TOCS)* (2007).

[4] ARNOLD, O., MATUS, E., NOETHEN, B., WINTER, M., LIMBERG, T., AND FETTWEIS, G. Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs. *ACM Transactions on Embedded Computing Systems (TECS)* (2014).

[5] ASMUSSEN, N., VÖLP, M., NÖTHEN, B., HÄRTIG, H., AND FETTWEIS, G. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[6] BALKIND, J., LIANG, X., MATL, M., WENTZLAFF, D., MCKEOWN, M., FU, Y., NGUYEN, T., ZHOU, Y., LAVROV, A., SHAHRAD, M., FUCHS, A., AND PAYNE, S. OpenPiton: An open source manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[7] BALLESTEROS FRANCISCO J., EVANS NOAH, F. C., AND GUARDIOLA GORKA, MCKIE JIM, MINNICH RON, S.-S. E. NIX: A case for a manycore system for cloud computing. *Bell Labs Technical Journal* (2012).

[8] BARBALACE, A., ILIOPOULOS, A., RAUCHFUSS, H., AND BRASCHE, G. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)* (2017).

[9] BARBALACE, A., LYERLY, R., JELESNIANSKI, C., CARNO, A., CHUANG, H.-R., LEGOUT, V., AND RAVINDRAN, B. Breaking the boundaries in heterogeneous-ISA datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[10] BARBALACE, A., RAVINDRAN, B., AND KATZ, D. Popcorn: a replicated-kernel OS based on Linux. *Ottawa Linux Symposium (OLS)* (2014).

[11] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)* (2009).

[12] BAUMANN, A., HAWBLITZEL, C., KOURTIS, K., HARRIS, T., AND ROSCOE, T. Cosh: Clear OS data sharing in an incoherent world. In *2014 Conference on Timely Results in Operating Systems (TRIOS)* (2014).

[13] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[14] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. The Gem5 simulator. *SIGARCH Computer Architecture News* (2011).

[15] BOHNENSTIEHL, B., STILLMAKER, A., PIMENTEL, J., ANDREAS, T., BIN LIU, TRAN, A., ADEAGBO, E., AND BAAS, B. A 5.8 pJ/Op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *IEEE Symposium on VLSI Circuits (VLSI-Circuits)* (2016).

[16] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of linux scalability to many cores. *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)* (2010).

[17] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM TOCS* (2015).

[18] COCK ET AL. Enzian: a research computer for datacenter and rackscale computing. In *Poster proceedings of the 13th European Conference on Computer Systems (EuroSys)* (2018).

[19] DAVID GOLUB, R. D., GOLUB, D., DEAN, R., FORIN, A., AND RASHID, R. Unix as an application program. In *In USENIX 1990 Summer Conference* (1990), pp. 87–95.

[20] DENNIS, J. B., AND VAN HORN, E. C. Programming semantics for multiprogrammed computations. *Communications of the ACM* (1966).

[21] DONGARRA, J. Report on the Tianhe-2A system. Tech. rep., University of Tennesssee Oak Ridge National Laboratory, 2017.

[22] FÄHNDRICH, M., AIKEN, M., HAWBLITZEL, C., HODSON, O., HUNT, G., LARUS, J. R., AND LEVI, S. Language support for fast and reliable message-based communication in singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)* (2006).

[23] FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. Microkernels meet recursive virtual machines. In *Proceedings of the second USENIX symposium on Operating systems design and implementation (OSDI)* (1996).

[24] GE, Q., YAROM, Y., CHOTHIA, T., AND HEISER, G. Time protection: the missing OS abstraction. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)* (2019).

[25] GERBER, S. *Authorization, Protection, and Allocation of Memory in a Large System*. PhD thesis, ETH Zurich, 2018.

[26] HAIBO, S. B.-W., RONG, C., YANDONG, C., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., AND WU, M. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2008).

[27] HARDY, N. KeyKOS architecture. *SIGOPS Operating Systems Review* (1985).

[28] HARRIS, T. Hardware trends: Challenges and opportunities in distributed computing. *ACM SIGACT News* (2015).

[29] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., AND SCHÖNBERG, S. The performance of μ-kernel-based systems. In *Proceedings of the sixteenth ACM symposium on Operating systems principles - (SOSP)* (1997).

[30] HEISER, G., AND ELPHINSTONE, K. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems (TOCS)* (2016).

[31] HP LABS. The Machine. https://www.labs.hpe.com/the-machine, 2018. Accessed: May, 2018.

[32] JÄRVINEN, K., AND SKYTTÄ, J. High-speed elliptic curve cryptography accelerator for Koblitz curves. In *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2008).

[33] KAESTLE, S., ACHERMANN, R., HAECKI, R., HOFFMANN, M., RAMOS, S., AND ROSCOE, T. Machine-aware atomic broadcast trees for multicores. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI)* (2016).

[34] KARNAGEL, T., HABICH, D., AND LEHNER, W. Adaptive work placement for query processing on heterogeneous computing resources. In *Proceedings of Very Large Data Bases (VLDB)* (2017).

[35] Kim, S., Huh, S., Zhang, X., Hu, Y., Wated, A., Witchel, E., and Silberstein, M. GPUnet: Networking abstractions for GPU programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[36] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)* (2009).

[37] Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Da Silva, D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V. K42: Building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys)* (2006).

[38] Krill, B., Amira, A., and Rabah, H. Generic virtual filesystems for reconfigurable devices. *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)* (2012).

[39] Kumar, A. Intel's new mesh architecture: The "superhighway" of the data center – IT Peer Network, 2017.

[40] Lackorzynski, A., and Warg, A. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems (IIES)* (2009).

[41] Liedtke, J. On µ-kernel construction. In *Proceedings of the fifteenth ACM symposium on Operating systems principles (OSDI)* (1995).

[42] Lin, F. X., Wang, Z., and Zhong, L. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)* (2014).

[43] Liu, D., Chen, T., Liu, S., Zhou, J., Zhou, S., Teman, O., Feng, X., Zhou, X., and Chen, Y. PuDianNao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015).

[44] Lyons, A., McLeod, K., Almatary, H., and Heiser, G. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)* (2018).

[45] Martin, M. M. K., Hill, M. D., and Sorin, D. J. Why on-chip cache coherence is here to stay. *Communications of the ACM (CACM)* (2012).

[46] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM (CACM)* (1960).

[47] McKenney, P., Appavoo, J., Kleen, A., Krieger, O., Russell, R., Sarma, D., and Soni, M. Read-copy update. *Ottawa Linux Symposium (OLS)* (2001).

[48] Miller, M. S., Yee, K.-P., Shapiro, J., et al. Capability myths demolished. Tech. rep., Johns Hopkins University Systems Research Laboratory, 2003.

[49] Needham, R. M., and Walker, R. D. The cambridge CAP computer and its protection system. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles (SOSP)* (1977).

[50] Nightingale, E. B., Hodson, O., McIlroy, R., Hawblitzel, C., and Hunt, G. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)* (2009), SOSP.

[51] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[52] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. Plan 9 from Bell Labs. In *Proceedings of Computing Systems, Volume 8* (1995).

[53] Reese, W. Nginx: the high-performance web server and reverse proxy. *Linux Journal 2008*, 173 (2008), 2.

[54] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. Ptask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[55] Shan, Y., Huang, Y., Chen, Y., Zhang, Y., and Osdi, I. LegoOS : A disseminated , distributed OS for hardware resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2018).

[56] Shapiro, J. S., Smith, J. M., and Farber, D. J. EROS: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP)* (1999).

[57] Silberstein, M. OmniX: an accelerator-centric OS for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)* (2017).

[58] Silberstein, M., Ford, B., Keidar, I., and Witchel, E. GPUfs: Integrating a file system with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[59] So, H. K.-H., and Brodersen, R. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transaction of Embedded Computing Systems (TECS)* (2008).

[60] Sodani, A. Knights landing (KNL): 2nd generation Intel® Xeon Phi processor. In *Proceedings of Hot Chips 27 Symposium (HCS)* (2015).

[61] Song, X., Chen, H., Chen, R., Wang, Y., and Zang, B. A case for scaling applications to many-core with OS clustering. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)* (2011).

[62] Steinberg, U., and Kauer, B. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)* (New York, NY, USA, 2010), ACM, pp. 209–222.

[63] Tanenbaum, A., Mullender, S., and Renesse, R. V. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)* (1986).

[64] Watson, R. N., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R., Roe, M., Son, S., and Vadera, M. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[65] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical capabilities for UNIX. In *USENIX Security Symposium (USENIX Security)* (2010).

[66] Wentzlaff, D., and Agarwal, A. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review* (2009).

[67] Woodruff, J., Watson, R. N. M., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R., and Roe, M. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)* (2014).

[68] Zellweger, G., Gerber, S., Kourtis, K., and Roscoe, T. Decoupling cores, kernels, and operating systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).