Doctoral Thesis

Authorization, Protection, and Allocation of Memory in a Large System

Author(s):
Gerber, Simon

Publication Date:
2018

Permanent Link:

https://doi.org/10.3929/ethz-b-000296835 →

DISS. ETH NO. 25300

Authorization, Protection, and Allocation of Memory in a Large System

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

SIMON GERBER

Master of Science ETH in Computer Science, ETH Zurich

born on 01.09.1986

citizen of Langnau i.E., Bern

accepted on the recommendation of

Prof. Dr. Timothy Roscoe (ETH Zurich), examiner
Prof. Dr. Gustavo Alonso (ETH Zurich), co-examiner
Dr. Robert N. M. Watson (University of Cambridge), co-examiner

2018

Abstract

In this dissertation, I rethink how an OS supports virtual memory. Classical virtual memory is an opaque abstraction of RAM, backed by demand paging. However, most systems today (from phones to data-centers) do not page, and indeed may require the performance benefits of non-paged physical memory, precise NUMA allocation, etc. Moreover, MMU hardware is now useful for other purposes, such as detecting page access or providing large page translation. Accordingly, the venerable VM abstraction in OSes like Windows and Linux has acquired a plethora of extra APIs to poke at the policy behind the illusion of a virtual address space.

Instead, I present Barrelfish's memory system which inverts this model. Applications explicitly manage their physical RAM of different types, and directly (though safely) program the translation hardware. Barrelfish's memory system requires no virtualization support, and outperforms VMM-based approaches for all but the smallest working sets. We show that Barrelfish enables use-cases for virtual memory not possible in Linux today, and other use-cases are simple to program and on par with Linux's performance.

Finally, I show how Barrelfish's capability system allows our memory model to scale to multiple cores. We present a set of algorithms which allow Barrelfish to process capability operations when capabilities exist on multiple cores without risking that different cores have different views of the global set of capabilities. The usual capability operations are sufficient to allow

our memory model to work on multiple cores, as I implement all the memory model primitives as capability operations. We demonstrate that the capability operations retain relatively low-latency in the presence of capabilities which exist on multiple cores.

Zusammenfassung

In dieser Dissertation überdenke ich die Abstraktionen von virtueller Speicherverwaltung, wie sie vom Betriebssystemen angeboten wird. Klassische Abstraktionen sind nicht mehr zeitgemäss, da sich die Hardwarelandschaft stark verändert hat. Virtuelle Speicherverwaltung dient heute nicht mehr primär zur Auslagerung von Speicher sondern um Performancevorteile von präzieser NUMA-Allokation auszunutzen und direktem Zugriff auf physikalischen Speicher zu ermöglichen, wie es von vielen Anwendungen von Smartphones bis hin zu Rechenzentren benötigt wird. Die Adressübersetzungshardware kann heute für viele weitere Zwecke genutzt werden. Zum Beispiel kann sie Zugriff auf bestimmte Adressen gewähren oder grössere Übersetzungseinheiten ("Seiten") anbieten. Klassische Betriebssysteme wie Windows und Linux bieten eine Vielzahl von Programmierschnittstellen an, um diese Funktionalität an Applikationen weiterzuleiten, so dass diese ihren virtuellen Adressraum ihren spezifischen Anforderungen gemäss anpassen können.

Stattdessen beschreibt diese Dissertation ein neues Modell für Speicherverwaltung. In diesem neuen Modell können Applikationen direkt Arbeitsspeicher allozieren und ihren virtuellen Addressraum verwalten, indem sie selbständig die Übersetzungshardware direkt programmieren. Das Betriebssystem stellt sicher, dass Applikationen nur Arbeitsspeicher, auf welchen sie Zugriff haben, in ihren Addressraum einfügen können. Wir beschreiben die

Implementierung des Modells in Barrelfish und zeigen, dass Applikationen virtuelle Adressierung für Zwecke brauchen können, welche in einem klassischen Betriebssystem nahezu unmöglich sind. Ferner zeige ich, dass andere Nutzungsfälle sowohl einfach zu programmieren als auch wettberwerbsfähig im Vergleich zu Linux sind.

Schliesslich beschreibt diese Dissertation wie das Capability-System von Barrelfish es erlaubt, mein Speicherverwaltungsmodell auf mehrere Prozessorkerne zu skalieren. Wir beschreiben die Algorithmen, welche es Barrelfish erlauben, Capability-Operationen auszuführen auch wenn Capabilities auf mehreren Kernen existieren. Die üblichen Capability-Operationen genügen, um unser Speicherverwaltungsmodell zu skalieren, da ich alle grundlegenden Operationen als Capability-Operationen implementiere. Zudem zeige ich, dass die Operationen eine relativ kleine Latenz beibehalten, auch wenn Capabilities auf mehreren Kernen existieren.

Acknowledgments

First, I would like to thank my thesis advisor, Timothy Roscoe, for offering me the opportunity to work on operating system research over the last six years, and for all the feedback and insightful discussions we have had on memory and resource management. Next I would like to thank Gustavo Alonso for agreeing to co-supervise my thesis, and for all the general feedback on presenting my research throughout my time in the Systems Group. Robert Watson gets a huge thank you for agreeing to be part of my thesis committee and for the valuable feedback on the weak spots of my dissertation. For improving the dissertation contents, special thanks go to Alan Cox, who has graciously explained the finer details of FreeBSD's superpage support.

Naturally, I would like to thank the whole Systems Group, past and present, for their support and great conversations over lunch and coffee. In particular, I would like to thank Simonetta, Jena, Eva, and Nadia, for the great work in taking care of us students with regards to the more tedious administrative issues.

Of course, I am grateful for the support that I was offered by my family throughout the years. Apart from my family, thanks also go to the current and past members of my shared flat. Thank you for putting up with me throughout paper deadlines etc.

I would like to specially mention all the Barrelfish team members past and present and all the students working on Barrelfish over the years without whom this thesis would not have been possible. In particular, I would like to thank "my generation" of the Barrelfish team – Kornilios, Stefan, Pravin, Gerd, Reto, Roni, Moritz, David, and Lukas – for their work and collaboration on all areas of Barrelfish. Also, of the students who have worked on Barrelfish, I would particularly like to thank Mark Nevill, who laid the groundwork for the distributed capability system, and often acted as a sounding board for the harder-to-debug issues during the time we lived together.

Last, but not least, I would like to thank Dejan Milojicic for the opportunity to visit HP labs in Palo Alto (now HPE labs) for an internship, during which I ended up working on Barrelfish in the context of HPE's TheMachine project. I would also like to thank the researchers, staff, and fellow interns at HP labs for the great time I had there.

Zurich, September 2018.

Contents

1

Introduction

This dissertation presents the design and implementation of a memory system for the *multikernel* operating system architecture.

My goal is to provide a memory system that

1. is *transparent* in regard to translation hardware features, that is, enables applications to utilize specific features of translation hardware without compromising on the design in the interest of performance,
2. is *scalable* with an increasing number of processor cores, and
3. provides a simple and orthogonal interface that avoids feature interactions and performance anomalies stemming from such feature interactions.

During the research for this dissertation, an implementation of this memory system has been created in the Barrelfish research operating sys-

tem [BBD+09]. In this dissertation, I use that implementation to argue the following thesis:

> An operating system's memory system can achieve the goals of scalability and transparency by letting applications directly manage physical memory, directly (but safely) program available translation hardware to build the environment in which they operate, by having the operating system reflecting virtual memory-related processor exceptions back to the faulting process, thereby essentially turning the classical virtual memory system inside out.
> Additionally, such an inverted memory system can achieve performance which is competitive with established memory systems, such as Linux, while avoiding the pitfalls of mechanism redundancy, policy inflexibility and feature interaction in the API presented to applications.

A very diverse set of applications can benefit from this design. This includes many server-class applications such as databases, language run-times which already build their own memory systems on top of the operating system abstractions, as well as other desktop and server applications that require specific data placement or translation granularities.

Motivation

As hardware manufacturers have battled with providing exponential increases in single-core performance for the last two decades, but the amount of transistors on a chip has continued to double every couple years, many manufacturers have made improvements to the raw processing power of a

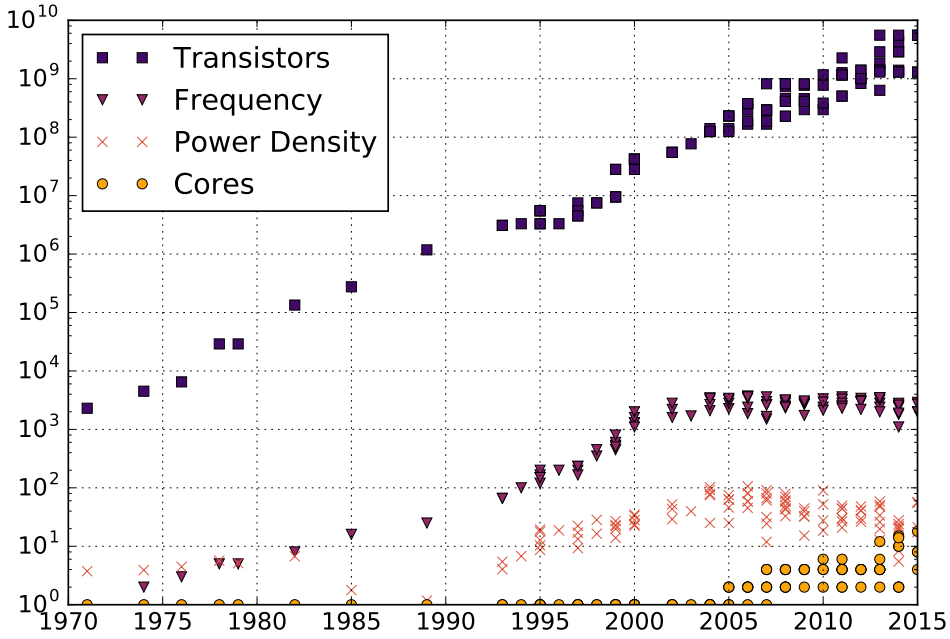


Figure 1.1: *Moore's law illustrated with data from various Intel processors [Wal]*

chip by providing more concurrent threads of execution, or *cores*, on a single chip. This is commonly called "multi-core".

Figure 1.1 illustrates the trend of stagnating single-core performance and increase in core count by plotting the number of transistors, clock frequency, power density, and number of cores for various Intel processors starting from the earliest Intel designs from the 1970s all the way to modern 20-core Xeon designs from 2015.

Managing and exploiting the available parallelism in current hardware is one of the main challenges for the development of software systems, as software needs to be scalable in order to exploit hardware parallelism. In this context, I define *scalability* as the ability of software to provide speedups proportional to the number of available threads of execution.

In particular, as the operating system sits between the hardware and appli-

cation software, it is an important piece of software when striving to make our software stacks scalable. Thus services offered by the OS, such as memory management, need to be scalable, in order to not prevent application software from scaling.

Alongside the trend to multi-core CPUs the landscape of memory technologies has changed significantly over the last few years, becoming significantly more heterogeneous. Today there are many emerging non-volatile random access memory (NVM) technologies which promise near DRAM latencies. One of the more promising NVM technologies is phase-change memory, which is the technology behind Intel and Micron's 3D Xpoint memory.

I argue that applications for modern machines should manage physical RAM explicitly and directly program MMUs according to their needs, rather than manipulating such hardware implicitly through a virtual address abstraction as in Linux. Previous work shows that for applications like databases the performance gains from closely managing the MMU mappings and locations of physical pages on memory controllers are as important to the end user as the functional correctness of the program [GARH14, LBKN14].

However, traditional virtual memory (VM) systems present a conceptually simple view of memory to the application programmer: a single, uniform virtual address space which the OS transparently backs with physical memory. In its pure form, applications never see page faults, RAM allocation, address translation, TLB misses, etc.

Naturally, this simplicity has a price. VM is an illusion — one can exhaust physical memory, resulting in thrashing, or the OS killing the application. Moreover, the performance of the VM abstraction is unpredictable, partially due to the fact that VM hardware is complex, with multiple caches, TLBs, page sizes, NUMA nodes, etc.

Consequently, the once-simple virtual memory (VM) abstraction in systems

such as Linux has become steadily more complex, as application developers demand more control over the mapping hardware, by piercing the VM abstraction with features like transparent huge pages, NUMA allocation, pinned mappings, etc. In section 2.2, I discuss the complexity, redundancy, and feature interaction in the formerly simple VM interface.

In this dissertation I show that explicit primitives for managing physical memory and the MMU deliver comparable or better application performance, greater functionality, and a simpler and orthogonal interface that avoids the feature interaction and performance anomalies seen in Linux.

For all of these reasons, I argue that classical VM is outdated, and hinders server-class applications more than it helps them.

Thus, this dissertation tries to answer the question of how to best manage memory in a modern system without hindering applications from achieving the best possible performance on a given hardware platform.

In response to the evolving demands of applications, I investigate the consequences of turning the VM system inside-out: applications (1) directly manage physical RAM, and (2) directly (but safely) program MMUs to build the environment in which they operate.

My contribution is a comprehensive memory system design which achieves these goals, allows applications to take full advantage of the features that are available in translation hardware, scales to large multicore machines, and which performs well.

I present an implementation of my design in Barrelfish. Barrelfish's memory system adopts a radically inverted view of memory management compared with a traditional system like Linux. Barrelfish processes use capabilities to manage physical RAM without requiring that RAM to ever be mapped in their address space. Nevertheless, Barrelfish processes still run inside a virtual address space (the MMU is enabled) but this address space is

securely constructed by the application itself with the help of a library which exposes the full functionality of the MMU through the capability system. Above this, all the functionality of a traditional OS memory system is provided as a library which applications can link against if they want to continue using traditional APIs such as the C standard library's venerable malloc and free.

Application-level management of the virtual address space is not a new idea. Earlier systems that provide application-level management of the virtual address space include the Exokernel system [EKO95], the V++ Cache Kernel [CD94], and more recently seL4 [KEH+09]. I review those systems and more in section 2.5.

Similarly, allowing applications to directly manage physical RAM has been previously proposed in the context of capability systems, such as KeyKOS [RHB+86], Hydra [CJ75, LCC+75, WLH81], and more recently – and the largest inspiration for Barrlefish's capability system – seL4 [KEH+09]. I review those systems and others which are pertinent to my design in section 2.3.

Contribution

In this dissertation, I make three main contributions:

1. A comprehensive design and implementation of application-level memory management for modern hardware capable of supporting applications which exploit its features. I extend the Barrelfish model to support safe user construction of page tables, arbitrary super-page mapping, demand paging, and fast access to page status information without needing virtualization hardware.

2. A detailed performance evaluation of Barrelfish's memory system comparing it with a variety of techniques provided by, and different configurations of, a modern Linux kernel, showing that useful performance gains are achieved while greatly simplifying the interface.

3. A rigorous design for decentralized capability management which enables scaling the Barrelfish memory system to multiple cores without impacting the safety guarantees made by the memory system or the capability system.

In particular, my personal contributions in the implementation are comprised of:

1. A mechanism ("mapping capabilities") which connects Barrelfish's memory and capability systems, which is described in chapter 3.

2. Support for advanced MMU features, such as large pages and changing page protections without deleting and recreating the mappings.

3. A working implementation of the distributed capability system which was first presented in Mark Nevill's master's thesis [Nev12].

Further personal contributions are the concept of turning the memory system inside out by utilizing various pieces of previous work that have never been combined in this fashion in a single system, and the evaluation of both the memory and capability system.

Structure of the Dissertation

The rest of this dissertation is structured as follows: In chapter 2, I give an overview and critique of the Linux and Windows memory management

systems, and discuss previous work in both capabilities and non-traditional memory systems. In chapter 3, I discuss how my inverted memory management system can be implemented for a single core and compare and discuss my system's performance with a recent Linux kernel. Then in chapter 4, I discuss tradeoffs to be made for multi-node capability systems, and discuss the design, implementation, and performance of my protocol for decentralized capability operations. In chapter 5, I provide a simple formal model for the capability protocol and discuss the challenges in formally verifying a protocol of this size. Finally, I draw some conclusions, and give some ideas for directions of future research in chapter 6.

Related publications

The work presented in this dissertation is part of the Barrelfish research project, and therefore depends on and supports the work of others.

A full and up-to-date list of publications related to Barrelfish can be found on the official Barrelfish website, under `http://www.barrelfish.org/documentation#publications`.

Some of the work presented in this dissertation is published in various forms, and is listed here for reference:

[**BBD+09**] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim 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*, October 2009.

[**Nev12**] An Evaluation of Capabilities for a Multikernel. Mark Nevill. Master's thesis, ETH Zurich, May 2012.

[**Ger12**] Virtual Memory in a Multikernel. Simon Gerber. Master's thesis, ETH Zurich, May 2012.

[**GZA+15**] Not Your Parents' Physical Address Space. Simon Gerber, Gerd Zellweger, Reto Achermann, Kornilios Kourtis, Timothy Roscoe, Dejan Milojicic. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems*, HOTOS XV, 2015.

2

Background and related work

In this chapter I give some background on the address translation hardware in modern processors in section 2.1.

Then I give an overview over the virtual memory systems found in Linux, FreeBSD and Windows in section 2.2, together with a critique which points out some undesirable properties that I want to consciously avoid in the design of my system.

In section 2.3, I provide background on capability systems, some of which have had a large influence on Barrelfish's design. Additionally, I briefly discuss some hardware-supported and programming language capability systems in 2.4

In section 2.5, I review other systems which proposed non-traditional memory systems.

Finally, in section 2.6, I provide background on the multikernel model and its implementation in Barrelfish.

Modern Virtual Memory Hardware

Intel

As described in Intel's Software Developer's Manual [Int, Vol. 3, ch. 3], Intel's x86 (IA-32 and EM64T) architectures, hardware support for memory management, address translation, and memory protection is present in two forms: segmentation and paging. Segmentation provides isolation of code, data, and stack modules and is not optional. Paging provides a traditional demand-paged virtual memory system which can be used for isolation as well. However, unlike segmentation, paging can be disabled completely. Most operating systems choose not to do so, as it is hard to work with limited amounts of physical memory and no demand paging.

Segmentation

Memory segmentation works by dividing the processor's addressable memory space (the linear address space) into smaller protected address spaces, the segments. Thus memory addresses in the processor are logical addresses (also called far pointers) that consist of a segment selector and an offset. The segment selector is a unique identifier for the segment which contains an offset into the global descriptor table (GDT). Using that offset, the processor retrieves a segment descriptor that contains the base and size of the segment as well as the access rights and privilege level for that segment. The linear address is then computed by adding the offset of the logical address to the base of the segment.

If paging is disabled, linear addresses are directly mapped to the physical address space, i.e. the range of addresses that the processor can generate on its address bus.

There are several different usage models for segmentation. The most basic model is called basic flat segmentation and hides most of the segmentation system from the operating system and applications. In this model, applications and operating system have access to a contiguous unsegmented address space.

The next level of usage is called protected flat segmentation and differs from basic flat segmentation by having segment limits that restrict program access to the address range that can actually contain physical memory.

The usage model that makes full use of the capabilities of the segmentation hardware is called multi-segment model. In this model each application has its own set of segments which – if so desired – can be shared among several cooperating applications.

Paging

As multitasking systems usually define a linear address space that cannot be mapped directly to physical memory due to its size, demand paging ("paging") virtualizes the linear address space, thus producing the more familiar "virtual addresses". The virtualization of the linear address space is handled by the processor's paging hardware. Using paging we can simulate a large linear address space with a small amount of physical memory and some disk storage. Using paging, each segment is split into pages of 4 KiB in size that are either stored in physical memory or on disk. The operating system has to maintain a page directory and a set of page tables to keep track of all the pages in the system. When a program attempts to access a linear address, the processor uses the page directory and page tables to translate the (virtual) linear address into a physical address and uses the generated physical address to perform the actual memory access (cf. Figure 2.1).

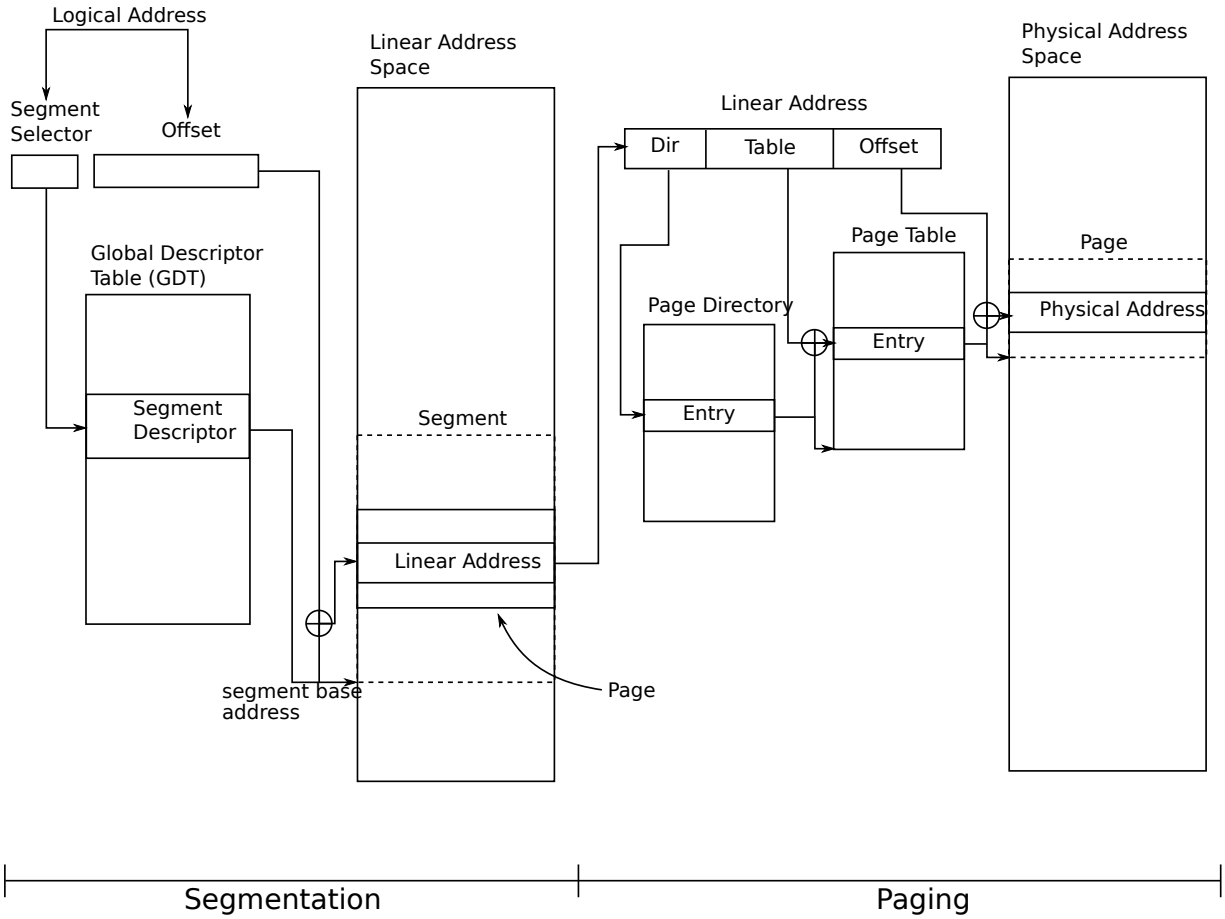


Figure 2.1: *Linear address lookup (from [Int, Vol.3A,p.3-2])*

If a page corresponding to a memory access (using a virtual address) is not in physical memory, the processor generates a page-fault exception, thus interrupting the program trying to access memory. The operating system then reads the missing page from disk (or allocates a new region of physical memory), installs that page in the appropriate page tables and resumes execution of the program.

As the paging mechanism described above is similar for 32 bit and 64 bit x86 processors, we have so far ignored the various subtle differences. In fact, there are three distinct paging models for x86 processors: standard 32-bit paging, PAE paging, and IA-32e paging.

Standard 32-bit paging uses 32-bit linear addresses and has a page directory

with 1024 entries that point to page tables containing 1024 page entries. Standard 32-bit paging can support physical page extension (PSE) that allows the physical addresses to be up to 40 bits wide. Standard 32-bit paging allows 4 KiB and 4 MiB pages.

PAE (physical address extension) paging is an extension of 32-bit paging that allows physical addresses to be 52 bits wide. When PAE is enabled, the processor maintains a set of four PDPTE registers that are loaded from a 32 byte page directory pointer table. These PDPTE registers are then used to translate linear addresses. Using PAE, all page table entries are 64 bits wide, and the system supports page sizes of 4 KiB and 2 MiB.

IA-32e paging is used on 64-bit processors and translates 48-bit linear addresses to 52-bit physical addresses. IA32-e uses four levels of page tables with 64-bit entries to translate addresses. IA32-e mode can support 4 KiB, 2 MiB, and 1 GiB pages. Of those page sizes, support for 4 KiB and 2 MiB pages is mandatory.

Table 2.1 gives an overview of the paging structures and their usage with the three paging modes.

Paging Structure	Paging Mode	Physical Address of Structure	Relevant Virtual Address Bits	Page Mapping[1]
PML4 Table	32-bit, PAE[2]	N/A		
	IA-32e[3]	CR3[4]	47:39	N/A
Page-directory Pointer Table (PDPT)	32-bit	N/A		
	PAE[2]	CR3[4]	31:30	N/A
	IA-32e[3]	PML4 entry	38:30	1 GB page[5]
Page directory	32-bit	CR3[4]	31:22	4 MB page[6]
	PAE[2], IA-32e[3]	PDPT entry	29:21	2 MB page
Page table	32-bit	PD entry	21:12	4 kB page
	PAE[2], IA-32e[3]		20:12	4 kB page

[1]This column specifies the size of leaf pages (if any) at this level of the page table tree. [2]PAE stands for Physical Address Extension, which extends physical addresses on a 32 bit processor from 32 to 52 bits. [3]IA-32e is short for IA-32 extended paging, which is the four-level paging mode on Intel 64. [4]CR3 is a special purpose register on x86 that contains the address to the current application's root page table. [5]Support for 1 GB pages is processor specific. Whether a processor supports 1 GB pages can be checked with the `cpuid` instruction. [6]Support for 4 MB pages on 32bit is processor specific and must be explicitly enabled in CR4.

Table 2.1: *Intel paging structures, from [Int, Vol. 3A, p.4-9]*

Caches

The Intel Software Developer's Manual [Int, Vol. 3, ch. 11] describes the memory cache and cache control mechanisms in Intel 64 and IA-32 processors. The sizes and characteristics of individual caches differ on different processor models and may change in future versions of the processor. Software can

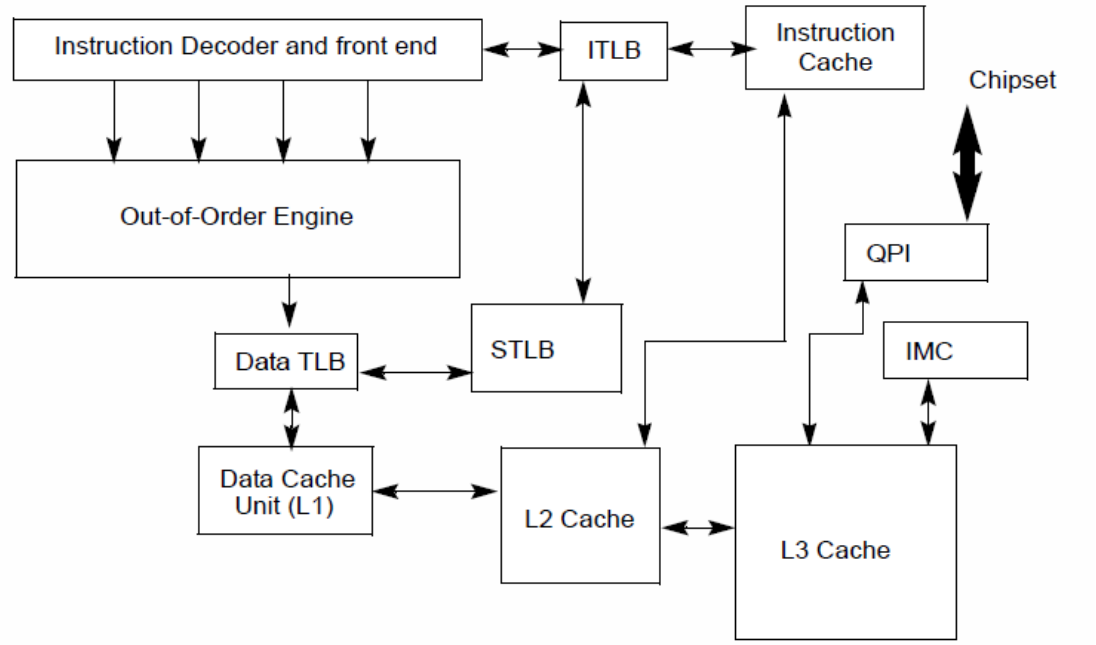


Figure 2.2: *Cache structure of the Intel Core i7 Processors [Int, Vol. 3, Figure 11-2].*

use the `cpuid` instruction to read sizes and characteristics of the caches for the processor on which the instruction is executed.

Generally speaking, Intel processors may implement four types of cache: the trace cache, the level 1 (L1) cache , the level 2 (L2) cache, and the level 3 (L3) cache. Whether a processor has these types of cache depends on the processor's family. In figure 2.2 the cache structure of the Intel Core i7 processors is displayed. In these processors, the L1 cache is divided into two sections: one section is dedicated to caching instructions and the other caches data. The L2 cache is a unified data and instruction cache. Each processor core has its own L1 and L2. The L3 caches is a unified data and instruction cache wihch is shared by all processor cores in a physical package. These processors do not implement a trace cache. The Intel SDN describes the cache structure for many processor families starting from the original Pentium on page 11–4 in volume 3A [Int].

Most modern Intel processors the cache line size for L1, L2, and L3 caches is 64 bytes. The processor will always reads a cache line from system memory beginning on a 64 byte boundary. A cache line can be filled from memory with a 8-transfer burst transaction. As the caches do not support partially-filled lines, caching any amount of memory requires caching an entire line.

Intel supports various memory types with different cacheabilities. These are listed in Table 11–2 in the Intel SDN. Some of the memory types are only available through the page attribute table (PAT) which extends the page table format to allow memory types to be assigned past the regular cache-disable and write-through bits in the page table entries.

Intel's processors implement the MESI protocol to maintain consistency of the L1 data and L2/L3 unified caches with caches of other processors.

TLB

The translation lookaside buffer (TLB) is used to cache the most recently used translation table entries. This speeds up memory accesses when paging is enabled by reducing the number of memory accesses that are required to read the page tables in order to complete a virtual to physical translation. On Intel processors, the TLBs are divided into four groups: instruction TLBs for 4 kB pages and large pages, and data TLBs for 4 kB and large pages. Processors based on Intel Core microarchitectures implement one level of instruction TLB and two levels of data TLB. Processors in the Core i3,i5, and i7 families provide a second-level unified TLB. Table 11–1 in the Intel SDN, volume 3 [Int] describes the size and associativity of each available TLB for each processor family. For our running example, the Core i7 family, the 4 kB page instruction TLB has 64 entries per hyperthread and is 4-way set associative, the large page instruction TLB has 7 entries

per thread and is fully associative. The 4 kB page data TLB has 64 entries, and the large page data TLB has 32 entries. Both first level data TLBs are 4-way set associative. The second level 4 kB page unified TLB has 512 entries and is 4-way set associative. Note that there is no second level large page TLB for Intel Core i7 processors.

ARMv7-A

ARMv7 defines its virtual memory architecture in the ARMv7 technical reference manual [ARM14, B2]. The memory system architecture of ARMv7-A is the *Virtual Memory System Architecture (VMSA)*. Additionally, ARMv7 defines multiple levels of caches, as well as allowing systems to have levels of caches beyond the ones defined in the techincal reference manual.

In this section, I will discuss the VMSA and the architecturally defined caches of ARMv7-A.

Virtual Memory System Architecture (VMSA)

The ARMv7-A Virtual Memory System Architecture (VMSA) supports translations of different granularities: *small pages* (4 kB) are the smallest unit of translation. The larger units of translation are: *large pages*, which translate 64 kB of memory, *sections*, which translate 1 MB of memory, and *supersections* which translate 16 MB of memory. Note that support for supersections is optional. The larger units of translation enable a large region of memory to be mapped using a single TLB entry.

The ARMv7-A VMSA defines a two level translation table. *First-level tables* hold first-level descriptors that contain a base address and either the translation properties of a section or supersection, or the translation properties and pointers to a second level table for large pages and small

pages. *Second-level tables* hold second-level descriptors each containing the base address and translation properties for a small page or a large page. Second-level tables are also referred to as *page tables*.

Large pages and supersections are special, because the top four bits of the page offset intersect with the bottom four bits of the page number used to look up the translation entry. Therefore, the VMSA requires that all sixteen table entries that refer to a single large page or supersection contain the same information.

ARMv7 is different from x86 paging, because the architecture defines two distinct translation table base registers (TTBR) and a translation table base control register (TTBCR) as opposed to the single x86 `cr3` register that holds the address of the current root table.

The normal use of the two TTBRs is that TTBR0 is typically used for process-specific addresses and TTBR1 is typically used for operating system and I/O addresses that do not change on a context switch.

The TTBR to use is determined by the most significant bits of the virtual address and the value of the `N` field of the TTBCR (TTBCR.N).

If either TTBCR.N is zero or the indicated bits of the virtual address are zero, TTBR0 is used. Otherwise TTBR1 is used.

The table pointed to by TTBR0 can range in size from 128 B to 16 kB depending on the value in TTBCR.N. If TTBCR.N is zero, all translations use TTBR0. The table pointed to by TTBR1 is always 16 kB in size. All first-level tables must be naturally aligned.

Second-level tables are 1 kB in size and must be naturally aligned. Each 32-bit entry in a table provides translation information for a 4 kB region of memory.

Caches

The description of architecturally defined caches in the ARMv7-A TRM [ARM14, A3.9] does not specify how to implement the cache hierarchy, because the details of such an implementation heavily depend on the microarchitecture. However, the TRM defines the application level interface to the memory system, and supports a hierarchical memory system with multiple levels of cache.

ARMv7-A defines a fairly comprehensive mechanism for managing hardware coherence of multiple caches, defining multiple different shareability domains. Additionally, ARMv7 supports both write-through, and write-back cacheable regions, where write-back regions can be either write-allocate or not.

TLB

The ARMv7 architecture [ARM14, B3.10] does not specify the exact form of the TLB structures for any design. Similarly to caches, the architecture only defines some principles for TLBs:

- The architecture has a concept of an entry locked down in the TLB. Implementations might not support lockdown.
- An unlocked entry in the TLB is not guaranteed to remain in the TLB.
- A locked entry in the TLB is guaranteed to remain in the TLB. However a locked entry in the TLB might be updated by subsequent updates to the translation tables.
- A translation table entry that returns a translation or access fault is guaranteed not to be held in the TLB. However an entry that returns a domain or permission fault might be held in the TLB.

- Any translation table entry that does not return a translation or access fault might be allocated to an enabled TLB at any time.
- TLB entries are not corrupted to give incorrect translations between disabling and re-enabling the MMU.

The ARMv7 VMSA allows the virtual memory map to be divided into global and non-global regions. Each non-global region has an associated address space identifier (ASID). ASIDs allow different translation table mappings to co-exist in a caching structure such as a TLB.

ARMv7 provides TLB maintenance operations which allow software to invalidate entries from a TLB. This operation is necessary for example when the operating system removes entries from a page table.

ARMv8-A

Virtual Memory System Architecture (VMSA)

ARMv8-A's virtual memory system architecture (VMSAv8) [ARM15, D4] is a evolution of ARMv7-A's VMSA. The biggest difference between the two VMSAs is that VMSAv8 has support for both one- and two-stage translations, that is, designed-in support for nested paging. Additionally, the VMSAv8 supports different translation granularities, i.e. different sizes for base pages. VMSAv8-64, the specification for a single-level translation of 64-bit virtual addresses, supports up to four levels of address lookup, input addresses of up to 48 bits, output addresses of up to 48 bits, and translation granularities of 4 kB, 16 kB, or 64 kB. For EL0 and EL1[1], VMSAv8 keeps

[1] ARMv8 has four exception levels which have decreasing rights. EL3 is the highest exception level and this is where ARM TrustZone firmware runs. EL2 is the next-highest exception level and designated for virtual machine monitors. EL1 is the exception level which is designated for the OS kernel, while applications run in EL0

two distinct TTBRs which are simply selected by inspecting the top bits of a virtual address. When the top bits are one, TTBR1 is used, otherwise TTBR0 is used. Notably, on ARMv8, all the registers associated with VMSAv8 are replicated for each exception level.

Caches

ARMv8 supports complex systems of caches [ARM15, D3.4]. The specification does not define exactly how to implement a cache hierarchy, but defines some features that any cache implementation must support. For example, the architecture has a concept of entries which are locked down in the cache. How to achieve cache lockdown is implementation defined, and lockdown might not be supported by a particular implementation, or some memory attributes which are supported by an implementation. The architecture guarantees that a locked entry remains in the cache. However it does not guarantee that such an entry remains dirty. Conversely, the architecture gives no guarantees for unlocked entries in the cache. Such entries might not remain in the cache and software must not assume that an unlocked item which remains in the cache remains dirty. The architecture has no mechanism that can guarantee that a memory location which is marked cacheable at the current or a higher exception level cannot be allocated to an enabled cache at any time. However, the architecture guarantees that a memory location that does not have a cacheable attribute cannot be allocated into the cache, and that memory locations which are not marked as cacheable in both the translation regime at the current exception level, and a translation regime at a higher exception level cannot be allocated to the cache. For data accesses, any memory location that is marked as "normal inner shareable" or "normal outer shareable" is guaranteed to be coherent with all masters in its shareability domain. Eviction of a cache entry from a cache level can overwrite memory that has been written by

another observer only if the entry contains a memory location that has been written to by an observer in the shareability domain of that memory location. Finally, the allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer if it was previously visible to that observer.

ARMv8 supports memory regions which are non-cacheable, write-through cacheable, or write-back cacheable. Additionally, ARMv8 defines the cache allocation hints read-allocate, transient read-allocate, no read-allocate, and write-allocate, transient write-allocate and no write-allocate. The cache transient hints provide a hint to the memory system that an access is non-temporal or streaming and unlikely to be repeated in the near future. The architecture does not require implementations to make use of cache allocation hints.

TLB

The ARMv8 architecture reference manual [ARM15] describes the architecture requirements in section D4.7.

The VMSAv8 supports TLB for each of its translation stages.

The principles which the architecture defines for TLBs are mostly identical with the ones given for ARMv7 TLBs earlier in this section. The changes are that ARMv8 does not have domain faults, and that TLB consistency is ensured when disabling and re-enabling a stage of translation rather than the whole MMU.

ARMv8 supports address space identifiers (ASID) for EL1 and EL0, but not EL2 or EL3. The architecture requires that ASID values are unique within any single inner shareable domain, that is each ASID value must have the same meaning to all processing elements in the system. The ASID size is

an implementation defined choice of 8 bits or 16 bits and can be queried by reading a memory model feature register.

ARMv8 defines a TLB maintenance instruction each for invalidating all entries in the TLB, invalidating a single TLB entry by ASID for a non-global entry, invalidate all TLB entries that match a specified ASID, and invalidate all TLB entries that match a specified VA regardless of the ASID. Each maintenance instruction can be specified as applying only to the processing element that executes the instruction or all processing elements in the same inner shareable shareability domain as the executing processing element.

Conclusion

Looking at three prominent hardware architectures for general-purpose processors, we see that the classical notion of virtual memory as an opaque abstraction of physical memory and the associated complexities has found its way into the translation hardware of both x86 and ARM-based processors.

The fact that the classical virtual memory model is assisted by hardware has led to a homogenization of the virtual memory systems of most modern operating systems, as I will discuss in the next section. However, as briefly discussed in the motivation, modern applications wish to control the placement and access latencies of their data and thus require controls that simply are not available in the classical virtual memory model. Thus, in the next section, I will outline and discuss the various holes that modern operating systems poke through the VM abstraction to accommodate applications.

Classical virtual memory

Unix was designed when RAM was scarce, and demand paging essential to system operation. Virtual memory is fully decoupled from backing storage via paging. Each process sees a uniform virtual address space. All memory is paged to disk by a single system-wide policy. The basic virtual memory primitive visible to software is `fork()`, which creates a complete copy of the virtual address space. Modern `fork()` is highly optimized (e.g. using copy-on-write).

Today, RAM is often plentiful, MMUs are sophisticated and featureful devices (e.g. supporting superpages), and the memory system is complex, with multiple controllers and set-associative caches (e.g. which can be exploited with page coloring).

Workloads have also changed. High-performance multicore code pays careful attention to locality and memory controller bandwidth. Pinning pages is a common operation for performance and correctness reasons, and personal devices like phones are often designed to not page at all.

Instead, the MMU is used for purposes aside from paging. In addition to protection, remapping, and sharing of physical memory, MMUs are used to interpose on main memory (e.g. for copy-on-write, or virtualization) or otherwise record access (such as the use of "dirty" bits in garbage collection).

In particular hardware support for translating larger pages has been targeted by previous research. Navarro et al. first proposed a mechanism for transparent operating system support for superpages in 2002 [NIDC02]. The key reason for this work was to take advantage of the increased TLB coverage provided by superpages. Correctly using superpages results in performance increases of over 30% in many cases. However, inappropriate use of superpages can result in enlarged application memory footprints, lead-

ing to higher pressure on physical memory and higher paging traffic. The increase of I/O cost associated with the paging traffic can easily outweigh any performance gains obtained by avoiding TLB misses.

At a high level, the design proposed by Navarro has the following components. Available physical memory is classified into contiguous regions of different sizes and is managed using a buddy allocator. A multi-list reservation scheme is used to track partially used memory reservations and is also employed to help choose reservations for preemption. A population map keeps track of memory allocations in each memory object, e.g. memory mapped files, and the code, data, stack and heap segments of processes. The system uses these data structures to implement allocation, preemption, promotion and demotion policies. External memory fragmentation is controlled by performing page replacements in a contiguity-aware manner. As the FreeBSD transparent superpage support was first introduced by this work, I will describe the implementation of transparent superpages in more detail in section 2.2.3.

Modern Linux

The need to exploit the memory system fully is evident from the range of features added to Linux over the years to "poke through" the basic Unix virtual address abstraction.

The most basic of these creates additional "shared-memory objects" in a process' address space, which may or may not be actually shared. Such segments are referred to by file descriptors and can either be backed by files or "anonymous". The basic operation for mapping such an object is `mmap()`, which in addition to protection information accepts around 16 different flags specifying whether the mapping is shared, at a fixed address, contains

pre-zeroed memory, etc. We describe basic usage of `mmap()` and related calls in Section 2.2.5; above this are a number of extensions.

Large pages: Modern MMUs support mappings at a coarser granularity than individual pages, typically by terminating a multi-level page table walk early. For example, `x86_64` supports 2 MB and 1 GB *superpages* as well as 4 kB pages, and for simplicity we assume this architecture in the discussion that follows (others are similar).

Today, Linux support for superpage mappings is somewhat complex. Firstly, mappings can be created for large (2 MB) or huge (1 GB) pages via a file system, `hugetlbfs` [Lina, Gor10a] either directly or through `libhugetlbfs` [Gor10b]. For each supported superpage size, a command-line argument tells the kernel to allocate a fixed pool of superpages at boot-time. This pool can be dynamically resized by an administrator. Shrinking a pool deallocates superpages from applications using a hard-wired balancing policy. In addition, one superpage size is defined as a system-wide default which will be used for allocation if not explicitly specified otherwise.

Once an administrator has set up the page pools, users can be authorized to create memory segments with superpage mappings, either by mapping files created in the `hugetlbfs` file system, or mapping anonymous segments with appropriate flags. Superpages may not be demand-paged [Azi14].

The complexity of configuring different memory pools in Linux at boot has led to an alternative, *transparent huge pages* (THP) [Linb, Cor14d]. When configured, the kernel allocates large pages on page faults if possible according to a single, system-wide policy, while a low-priority kernel thread scans pages for opportunities to use large pages through defragmentation. Demand-paging is allowed by first splitting the superpage into 4 kB pages [Azi14]. A typical modern `x86_64` kernel is configured for transparent support of 2 MB pages, but not 1 GB pages. Alternatively, an administrator

can disable system-wide THP at boot or by writing to sysfs and programs can enable it on a per-region basis at runtime using `madvise()`.

NUMA: The `mbind()` system call sets a NUMA policy for a specific virtual memory region. A policy consists of a set of NUMA nodes and a mode: *bind* to restrict allocation to the given nodes; *preferred* to prefer those nodes, but fall back to others; *interleaved* to interleave allocations across the nodes, and *default* to lazily allocate backing memory on the local node of the first thread to touch the virtual addresses. This "first touch" policy has proved problematic for performance [DFF+13].

`libNUMA` provides an additional `numa_alloc_onnode()` call to allocate anonymous memory on a specific node with `mmap()` and `mbind()`. Linux can move pages between nodes: `migrate_pages()` attempts to move all pages of a process that reside on a set of given nodes to another set of nodes, while `move_pages()` moves a set of pages (specified as an array of virtual addresses) to a set of nodes. Note that policy is expressed in terms of virtual, not physical, memory.

There are also attempts [Cor12c, Cor12a, Cor12b, Cor13a, DFF+13, Cor14b] to deal with NUMA performance issues transparently in the kernel, by migrating threads closer to the nodes containing memory they frequently access, or conversely migrating pages to threads' NUMA nodes, based on periodically revoking access to pages and tracking usage with soft page faults. A good generic policy, however, may be impossible; highly performance-dependent applications currently implement custom NUMA policies by modifying the OS [DFF+13].

User-space faults: Linux signals can be used to reflect page faults to the application. GNU libsigsegv [HB] provides a portable interface for handling page faults: a user fault handler is called with the faulting virtual address

and must then be able to distinguish the type of fault, and possibly map new pages to the faulting address. When used with system calls such as `mprotect()` and `madvise()`, this enables basic user-space page management. The current limitations of this approach (both in performance and flexibility) have led to a proposed facility for user-space demand paging [Cor13b, Cor14c].

Windows NT

When Windows NT was first designed in the early 1990s – the first version, Windows NT 3.1 was released in 1993 – RAM was scarce, and demand paging essential to system operation. Therefore the NT memory system is modelled closely after the traditional Unix VM model, where virtual memory is fully decoupled from backing storage via paging. However, just as modern Linux VM has various ways to poke holes into the VM abstraction, cf. section 2.2.1, NT has a number of ways in which application workloads which are sensitive to memory can tune how their virtual memory regions get backed.

Basic API and concepts: Memory management in Windows NT [YRSI17, Mar12] is built around the `VirtualAlloc*` family of API functions. This is the "Virtual API". On top of the Virtual API, Windows provides functions for small allocations – usually smaller than a page. Those functions are grouped into the "Heap API". The Heap API provides all the functions necessary to instantiate and make use of a memory heap. Alongside the Virtual API, Windows NT provides specialized functions for mapping files into an address space or sharing memory between processes. These API functions are grouped in the "File Mapping API".

Overall many of these concepts should look familiar. If we take the Virtual API and File Mapping API and write one API function for everything, we

end up with an API function that looks a lot like POSIX's `mmap` while the Heap API provides the same functionality as `brk` on a Unix system.

Shared memory: To allow processes to use the File Mapping API create shared memory regions, the NT kernel internally uses section objects which are exposed as file-mapping objects to the processes. Section objects are one of the fundamental primitives in the NT memory manager and are used to map virtual addresses to main memory, the page file[2], or some other file for which the application wants to create a memory-mapped window. Additionally, a section can be opened by one process or by many. Therefore, it would be an oversimplification to say that section objects equate to shared memory.

However, in the context of this brief overview of Windows memory management, we focus on how sections are used to provide shared memory to Windows processes. Section objects can be connected to committed memory to provide shared memory. Such a section object is called a page-file-backed section because pages connected to this section are written to the page file if pages linked to the section are evicted from physical memory. However, even if Windows is configured to run without a page file, we can still create page-file-backed sections, which then are backed only by pages in physical memory.

We can create a shared memory section by calling any of the `CreateFileMapping*` functions with `INVALID_HANDLE_VALUE` as the file handle. Optionally, we can provide a name and security descriptor for the new section. If we name the section, other processes can then open it by calling `OpenFileMapping` or `CreateFileMapping*` functions. Otherwise, a process can grant access to a section through handle inheritance by specifying that the handle to the

[2]Windows's terminology for swap space

section is inheritable when opening or creating the handle. Finally, we can also explicitly duplicate handles to section by calling `DuplicateHandle`.

Large pages: Before an applications on Windows NT can use large pages, a system administrator needs to configure the user account under which the application will run to have the `SeMemoryLockPrivilege` privilege, and each process that wishes to use large pages needs to enable the privilege in its process control block.

NT supports private large page mappings through `VirtualAllocEx` when the `flAllocationType` parameter is set to `MEM_RESERVE | MEM_COMMIT | MEM_LARGE_PAGES`. Because NT does not support paging out large pages, regions created with this method are not part of the process' working set which is otherwise used to determine which pages may be paged out.

It is also possible to create a "paging file backed section" to create a shareable memory region backed with large pages. This can be achieved by calling `CreateFileMapping` with parameter `flCommit` set to `SEC_COMMIT | SEC_LARGE_PAGES`. Internally, NT creates "virtual" last level page table entries for sections which are mapped with 2MB ranges in hardware. This is necessary because NT has two different concepts which are necessary when mapping a file or section into an address space. The *section* is the region or file as represented by one or more leaf tables. The *view* is the link between a process' page tables and the section's leaf page tables. Notably, views support an offset into a section, and the only restriction on the offset is that it is a multiple of 64kB on x64 Windows. This is the reason why NT creates virtual last level page table entries for sections that are backed with large pages. However, as soon as such a view with offset is created, the advantage of large pages is lost, as the large page mapping gets converted to regular 4kB pages, to enable mapping an arbitrary 64kB aligned subregion of the section.

FreeBSD

FreeBSD has a very standard Unix-style memory system which presents each process with a virtual address space which is managed by the BSD kernel. The implementation of the virtual memory system is based on the Mach 2.0 virtual memory system [Tev87], with updates from Mach 2.5 and 3.0. BSD adopted Mach's memory system because it features efficient support for sharing and a clean separation of machine-independent and machine-dependent features. FreeBSD uses *mmap* to provide shared memory both backed by files and anonymous regions backed by files in `tmpfs`. This is functionally identical to Linux's mmap shared memory regions.

Support for large pages: Navarro et al. use FreeBSD to demonstrate the benefits of using large pages [NIDC02]. The main FreeBSD implementation gained large page support in 8.0. Internally FreeBSD calls large pages "superpages", adopting the terminology used by Navarro. FreeBSD provides "transparent" support for superpages, where the kernel decides to use superpages without hints from the application. The kernel decides on the first page fault to a region of memory whether to create a "superpage reservation" or not. Anonymous regions, e.g. heap and stack, are always eligible for superpages because they often grow. However, mapped files must be at least of superpage size, because they grow much less often. On the first fault, the kernel may choose to reserve a superpage, but will only map a single 4 kB page. Additionally, the kernel keeps track of the offset of objects into a superpage, to allow sharing of superpages between processes. Finally, superpages have population maps which track used pages in the superpage.

When a superpage reservation has faulted in every page in its reservation, it can be *promoted* to a superpage. At this point, the kernel needs to decide

whether to make the promoted page read-write or read-only. A superpage is only promoted read-write when every page in the superpage is modified. Otherwise, the superpage is promoted read-only and split back into small pages when writes happen. If all small pages of a read-only superpage are modified, that superpage is promoted to read-write. The kernel keeps cached and free pages on buddy lists (organized by number of adjacent pages) which can be used to aggregate small pages back into superpages.

When a superpage is selected for paging (or swapping) out by the page daemon[3], the superpage mapping is demoted and one of the 4 kB page mappings is destroyed, so that future accesses to the superpage may trigger promotion again. Individual 4 kB pages of the superpage that are accessed are moved back to the active queue, the remainder will sit on the inactive queue. The physical superpage is only broken when one of the 4 kB pages on the inactive queue is freed.

Apart from the promotion mechanism, FreeBSD will immediately create superpage mappings when the data is already present in a physical superpage, e.g. when mapping the text section of an executable a superpage is created without waiting for an access.

Solaris

Solaris supports large pages for its “intimate shared memory” (ISM), “dynamic intimate shared memory” (DISM), and starting in Solaris 11.3 “optimized shared memory” (OSM).

All these mechanisms are built as extensions or options to System V shared memory segments which are mapped using the *shmat* call.

[3]The page daemon is the FreeBSD kernel thread which is responsible for finding and clearing unreferenced page mappings

The original ISM requires that the full region which should use large pages is pinned in main memory. DISM relaxes this requirement and allows pages to be swapped out, but requires that the swap space is large enough to accommodate the full DISM region. Finally, OSM is a new interface which is similar to *shmget* but takes an additional parameter called "granule_size" which is a power of two greater or equal to the system's configured page size. The size of the requested region must be a multiple of the granule size. The granule size is then the unit of operation on the OSM region. The region must be mapped aligned to the granule size, and any operations on the region, such as *madvise* calls, must be made on a granule size aligned boundary.

Initially an OSM region will not be backed by anything. To back a range inside an OSM region, the application needs to "lock" the range, after which any parts of the range that were previously unlocked will be filled with zero and the whole range will be accessible. If a range is no longer needed, it can be "unlocked" to relase the backing memory to the system.

Discussion

Based on the simple Unix virtual address space, the Linux VM system has evolved in response to new demands by accreting new features and functionality. This has succeeded up to a point, but has resulted in a number of problems.

The first is **mechanism redundancy**: there are multiple mechanisms available to users with different performance characteristics. For example, Figure 2.3 shows the performance of three different Linux facilities for creating, destroying, and changing "anonymous mappings": regions of virtual address space backed by RAM but not corresponding to a file. These

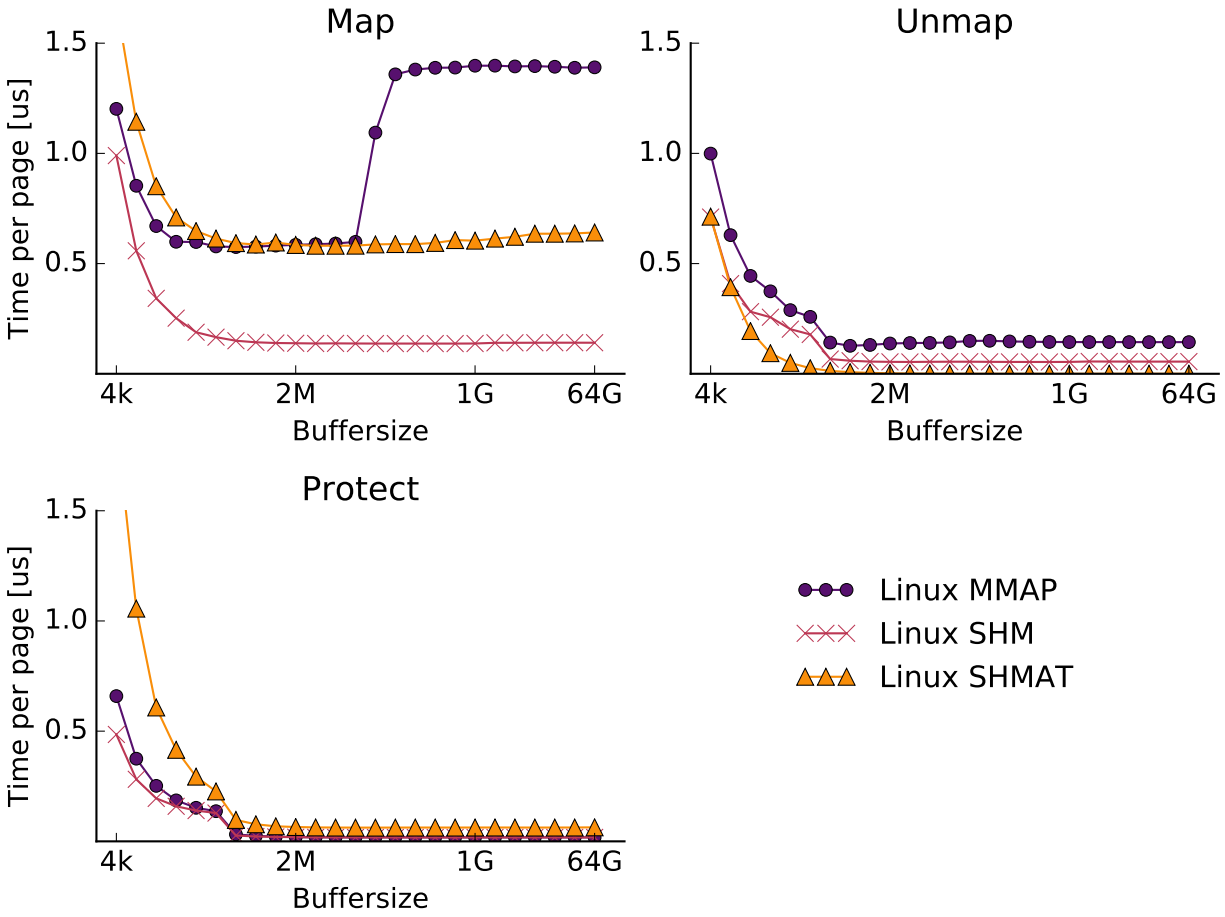


Figure 2.3: *Linux large page API comparion (4.2.0)*

measurements were obtained using the machine in Table 2.2 using 4k pages throughout.

MMAP uses an `mmap()` call with `MAP_POPULATE` and `MAP_ANONYMOUS` to map and unmap regions, and `mprotect()` for protection. This forces the kernel to zero pages being mapped, dominating execution time. Avoiding this behavior, even when safe, requires kernel reconfiguration at build time – a global policy aimed at embedded systems.

SHM creates a shared memory object with `shm_open()` and passes it to `mmap()` and `mprotect()`. In this case, `mmap()` will not zero the memory. Unmapping is also faster since memory is not immediately reclaimed. The

object can be shared with other processes, but (unlike MMAP mappings) cannot use large pages.

SHMAT attaches a shared segment with `shmat()`, and *does* allow large pages if the process has the `CAP_IPC_LOCK` capability. Internally, the mechanism is similar to `mmap()`, with system-wide limits on the number and size of segments.

For buffers up to 2 MB, the cost per page decreases with size for all operations due to amortization of the system call overhead. Afterwards, the time stays constant except for MMAP map operations.

`libhugetlbfs` provides `get_hugepage_region` and `get_huge_pages` calls to directly allocate superpage-backed memory using a `malloc`-style interface. The actual page size cannot be specified and depends on a system-wide default; 4 kB pages may be used transparently unless the `GHR_STRICT` flag is set. By default, `hugetlbfs` prefaults pages.

The high-level observation is: *No single Linux API is always optimal, even for very simple VM operations.*

A second problem is **policy inflexibility**. While the appropriate policy for many memory management operations such as page replacement, NUMA allocation or handling of superpages depend strongly on individual application's workloads. In Linux, however, they usually either apply system-wide, require administrator configuration (often at boot), must be enabled at compile time, or a combination of them.

For example, supporting two superpage sizes in `hugetlbfs` requires two different, pre-allocated pools of physical memory, each assigned to a different file system, precluding a dynamic algorithm that could adapt to changing workloads.

In addition to the added complexity in the kernel [Cor14a], the system-wide policies in *transparent* superpage support have led to a variety of performance

CPU	Intel Xeon E5-2670 v2 (Ivy Bridge)
#nodes / #sockets / #cores	2 / 2 / 20 @ 2.5 GHz
L1 / L2 cache	32 kB / 256 kB (per core)
L3 size	25 MB (shared)
dTLB (4 kB pages)	64 entries (4-way)
dTLB (2 MB pages)	32 entries (4-way)
dTLB (1 GB pages)	4 entries (4-way)
L2 TLB (4K)	512 entries (4-way)
RAM	256 GB (128 GB per node)
Linux kernel	v.4.2.0 (Ubuntu 15.10)

Table 2.2: *Test bed specifications. [Int14]*

`4.2.0`	4.2.0 (Ubuntu 15.10)	No large page support
`4.2.0-tlbfs`	4.2.0 (Ubuntu 15.10)	`hugetlbfs` enabled
`4.2.0-thp`	4.2.0 (Ubuntu 15.10)	Transparent huge pages enabled
`3.16`	3.16	Stock 3.16 kernel
`3.16-dune`	3.16	Linux 3.16 with Dune

Table 2.3: *Tested Linux configurations*

issues: Oracle DB has suffered from I/O performance degradation when reading large extents from disk [Cas13, Azi14]. Redis incurs unexpected latency spikes using THP due to copy-on-write overhead for large pages, since the application periodically uses `fork()` to persist database snapshots [San]. The `jemalloc` memory allocator experiences performance anomalies due to its use of `madvise` to release small regions of memory inside of bigger chunks which have been transparently backed by large pages — the resulting holes preventing later merging of the region back into a large page [Eva15].

These issues are not minor implementation bugs, but arise from the philosophy that memory system complexity should be hidden from applications,

and resource allocation policies should be handled transparently by the kernel.

The third class of problem is **feature interaction**. We have seen how superpages cannot be demand paged (even though modern SSDs can transfer 2MB pages with low latency). Another example is the complex and subtle interaction between kernel-wide policies for NUMA allocation with superpage support [Lina]. At one level, this shows up in the inability to control initial superpage allocation at boot time (superpages are always balanced over all NUMA nodes). Worse, Gaud et al. [GLD+14] show that treating large pages and NUMA separately does not work well: large pages hurt the performance of parallel applications on NUMA machines because *hot pages* are more likely, and larger, and *false page sharing* makes replication or migration less effective. Accordingly, the Carrefour [DFF+13] system modifies the kernel's NUMA-aware page placement to realize its performance gains.

While Windows NT looks somewhat different in the details, it is clearly also an evolution of the classical VM approach. Mechanism duplication seems to be less prevalent in the Windows API, as there are no obvious instances of it in the memory system. However, the Windows NT memory system also suffers from policy inflexibility and feature interaction, which is most obvious when creating a section view with an offset: this implicitly disables large pages for the section backing that view, even though it would be possible to associate the mappings with each view instead of the section itself. The underlying problem is that mapping granularity is mostly determined by a combination of theoretically unrelated policy choices which impact mapping granularity due to implementation choices.

In contrast to Linux or Windows NT, FreeBSD does not require superpages to be pinned, and allows paged out when memory demand is high. However, when a superpage is selected to be paged out it is broken up into its consistuent 4 kB pages. Additionally, applications on FreeBSD have less

control over page size than applications on both Linux and Windows NT, as FreeBSD does not offer any way to explicitly select a page size for a virtual region. The FreeBSD justification for this is that transparently selecting page size for applications leads to the best performance, which as discussed above in the context of Linux's THP is not always true [Cas13, Azi14, San, Eva15].

Collectively, these issues motivate investigating alternative approaches. As memory hardware diversifies in the future, memory management policies will become increasingly complicated. We note that none of the Linux memory APIs actually deal with *physical* memory directly, but instead select from a limited number of complex, in-kernel policies for backing traditional *virtual* memory.

In contrast, therefore, Barrelfish's memory system safely exposes to programs and runtime systems both physical memory and translation hardware, and allows libraries to build familiar virtual memory abstractions above this.

An overview of capability-based systems

Capability-based systems are one way of addressing the resource management problem. In the sixties and seventies, first approaches to address authorization with a variety of hardware and software techniques were proposed. We show a tabular overview of systems in Table 2.4. Those systems can be categorized into the three categories: hardware supported capabilities, kernel supported capabilities, and programming language systems. As this dissertation focuses on kernel supported capabilities, we further categorize kernel supported capability systems based on the mechanism they use to prevent unprivileged actors to gain access to capability metadata.

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 which 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, such as 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.[CKD94a] 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[CKD94a], 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 than 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 which is not directly 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 systems such as KeyKOS, EROSr, 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.[CPK08], 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.

We show a tabular overview of different systems which use capabilities in some form in table 2.4. In that table we label the systems with one of three categories: hardware supported capabilities, kernel supported capabilities, and programming language systems.

Kernel supported capabilities

Even without hardware support, capabilities can still be implemented in the OS kernel if all operations on capabilities have to be executed by the kernel on behalf of the applications.

One way to make capabilities unforgeable is to rely on sparsity in the capability key space or cryptographic methods which are the methods employed by operating systems such as Chorus [RAA+91] and Amoeba [MvRT+90].

Another way to protect capabilities is to make them kernel objects which can only be referred to by opaque handles from applications. Example systems that employ this strategy are KeyKOS [RHB+86], Hydra [CJ75, LCC+75, WLH81], EROS [SSF99], Mach [ABG+86, Iii91, FFB+88, Seb91] and Accent [RR81]. There exist some more recently developed systems such

Class	System	Capability features				
		Distribution	Persistency	Revocation; Garbage collect	Granularity	HW/SW support
Hardware, ISA Support	CAP	single process	yes, obj level	no; no	fine	ISA OS
	Plessey System 250	multi-node	yes	no; mem segments & caps	fine	ISA support, segm. VM
	StarOS	multi-node	no	no; GC across clusters	fine	OS supported
	IBM/38	single-node	yes	yes; no GC	fine	microcode OS, horiz/vert migr
	iAPX 432	multi-node	yes	no; no GC	fine	substantial ISA support
	Hardbound, low-fat ptrs hw	single-process	no	no; no GC	objs	ISA, compiler support
	CODOMs	single-node	no	yes; no GC	page + objs	HW/ISA support
	M-Machine	single-process	yes	yes; yes, GC VAS	fine	HW/ISA support
	CHERI	single-process	no	no; no	fine	ISA, OS compiler
Operating Systems	Hydra	multi-process	yes, obj level	no; yes, ref cnts and GC	fine	OS supported
	Mach, Chorus	multi-node	yes, cap to pager	yes; no GC	page	MMU, OS
	Amoeba	multi-node	yes, file system	no; no GC	page	Cryptography
	KeyKOS	multi-process	yes, VAS	no; no GC	objs	TCB/OS
	EROS	single-node	yes, obj level	yes; no GC	page + objs	MMU, OS
	L4	single-node	yes, cap to pager	OS level; memory + caps	page + objs	MMU, OS
	Barrelfish	multi-node	no	yes, OS level; No	page + objs	MMU, OS
	Composite	multi-node	no	yes; reference counts	page + objs	MMU, OS
Languages, fat pointers	Prog lang: E, Joe-E, Caja	single-process	yes, obj level	no; GC optional	objs	language runtime
	Softbound, CCured, low-fat ptrs sw	single-process	no	no; no GC	objs	compiler transformation
	Cyclone	single-process	no	no; GC optional	objs	compiler

Table 2.4: *Survey of related capability-based systems*

as some variants of L4 [HHL+97], e.g. seL4 [KEH+09] and L4Re [LW09], which also adopt kernel supported capabilities. L4Re and seL4 use kernel-protected capabilities to mediate access for both memory and objects. The kernel-maintained capability derivation tree allows for recursive revocation.

Kernel capabilities either provide memory protection at page level, or require kernel invocations for using smaller objects which discourages the use of kernel capabilities purely for protection reasons. Throughout this dissertation, we argue that kernel capabilities have benefits other than fine grained protection which makes them worth exploring in the context of a modern operating system.

Barrelfish's [BBD+09] capability system borrows heavily from seL4 and allows applications to only execute a set of safe operations on capabilities. The Barrelfish capability system is additionally designed as a distributed system which maintains the global set of capabilities in a machine across all Barrelfish CPU drivers running on individual cores in the machine. We give an overview over Barrelfish's architecture and the most relevant components in section 2.6.

However, Barrelfish employs kernel-supported capabilities to implement most of its kernel programming interface (KPI). This is achieved by extending the notion of typed capabilities as introduced by seL4 to provide a rich set of kernel object types with associated invocations. Today the Barrelfish KPI consists of twelve system calls, but only three of them are intended to be used in regular operation. These system calls are *INVOKE*, *YIELD*, and *LRPC*. Notably, the LRPC system call is not required but provides improved performance and a fast path through the kernel for LRPC-like[BALL90] inter-process messages where the sender of the message donates the rest of their scheduling time slice to the recipient. The rest of Barrelfish's KPI is implemented as capability invocations which can be thought of as method calls on a specific kernel object which is identified by a capability.

The set of possible invocations for each kernel object is defined by the object's capability type. Currently, Barrelfish's capability system knows of 50 distinct capability types, and a total of 84 distinct invocations, 34 of which are attached to the special *Kernel* capability type which is only held by the "monitor", which is the user-space component of the Barrelfish kernel. The *Kernel* capability grants the holder full access to the capability system. This is required for the monitor to manage the distributed shards of the capability derivation database. In chapter 4, we will present the protocol which is utilized to manage capabilities across cores.

A previous system which used capabilities in a distributed operating system is Amoeba [MvRT+90]. Amoeba is designed to tie together workstations, specialized servers, and a processor pool in a single distributed OS. To address shared resources in that environment, Amoeba uses sparse capabilities that are 128-bit values of which 48 bits are used for cryptographic protection to prevent users from tampering with the capabilities which they hold. Notably, Amoeba's resource management was orchestrated in a centralized fashion, where one of the specialized servers was responsible for creating and providing capabilities to all servers and clients in an Amoeba instance.

Other types of capabilities

We also give a quick overview of some capability systems which address a different problem, namely application-level pointer safety.

Hardware supported capabilities

Some early capability systems like CAP [NW77], StarOS [JCD+79], and IBM System/38 [HSH81] extend the processor's ISA with special instructions and registers which allow the hardware to enforce protection for even smallest objects without mediation of a trusted entity.

More recently, a number of systems were proposed that revived hardware capabilities. The focus of the M-Machine [CKD94b] system was a capability-system running in a single virtual address space. The M-Machine uses hardware-based guarded pointers, which allow the system designers to protect privileged system modules which exist in the single address space. The hardware support enable efficient checking and dereferencing of the 64-bit guarded pointers. Notably the M-Machine did not focus on being compatible with code that is not aware of capabilities.

In contrast, CHERI [WWN+15] retains compatibility with capability-unaware code. CHERI supports two types of capabilities. Regular CHERI capabilities are similar to the M-Machine guarded pointers but much larger, as the architecture represents capabilities in 256 bits. CHERI also supports *sealed* capabilities which allow software to construct an object capability system. The CHERI ISA includes instructions for hardware-assisted object invocations. Notably, CHERI still provides an MMU that supports traditional paging alongside its hardware-based capabilities. Thus a regular multi-address space operating system can employ CHERI capabilities to

provide fine-grained protection within a virtual address space, while sealed capabilities enable a form of inter-address space capability operations.

CODOMs [VBYN+14] is another recent system with hardware support for isolation between components. The CODOMs work presents a capability extension for x86 which is based on guarded pointers and provides low overheads, enables transparent integration, and does not require expensive memory tagging.

Programming language systems

Another area where capabilities, or capability-like entities, often appear are programming language run times. Some runtimes for object-oriented languages such as the JVM use some ideas that are fairly similar to capabilities to ensure referential correctness of programs by strictly separating memory areas holding references from areas holding data. Other languages provide capabilities more directly, for example languages such as E [Mil06], Joe-E [MW10] and Caja [MSL+08] rely on the compiler and language runtime to enforce a strict object-capability model.

While providing strong type safety, either using capability ideas or in general, is fairly straightforward in high-level languages with a rich language runtime, for lower-level languages this strong type safety is harder to achieve, however many approaches to prevent accessing data-structures outside of their bounds exist.

Software-based solutions include Softbound [NZMZ09], CCured [NMW02], Cyclone [JMG+02] and low-fat pointers (software variant) [DY16]. Apart from capability-like systems, there is work on hardware supported bounds-checking techniques like Hardbound [DBMZ08], Intel MPX [Int13] and low-fat pointers [KDS+13] that try to reduce the software overhead of supporting bounds checks in lower-level languages.

Non-traditional memory systems

In this section, I discuss previous work which presented non-traditional, i.e. not classical VM, memory systems.

Application-level memory management

The idea of moving memory management into the application rather than the kernel or an external paging server had been around for some time. Prior to Barrelfish and seL4, Engler et al. in 1995 [EGK95] outlined much of the motivation for moving memory management into the application rather than the kernel or external paging server, and described AVM, an implementation of application-level memory management for the Exokernel [EKO95] based on a software-loaded TLB, presenting a small performance evaluation on microbenchmarks. AVM referred to physical memory explicitly by address, and "secure bindings" conferred authorization to map it. Since then, software-loaded TLBs have fallen out of favor due to hardware performance trends. Both seL4 and Barrelfish target modern hardware page tables, and use capabilities to both name and authorize physical memory access.

The V++ Cache Kernel [CD94] implemented user-level management of physical memory through page-frame caches [HC92] allowing applications to monitor and control the physical frames they have, with a focus on better page-replacement policies. A virtual address space is a segment which is composed of regions from other segments called bound regions. A segment manager, associated with each segment, is responsible for keeping track of the segment to page mappings and hence handling page faults. Pages are migrated between segments to handle faults. Segment managers can run separately from the faulting application. It is critical to avoid double

faults in the segment manager. Initialization is handled by the kernel which creates a well-known segment.

Customizable policies

Other systems have also reflected page faults to user space. Microkernels like L4 [LUE+99], Mach [RTY+88], Chorus [ARS89], and Spring [KN93] allow server processes to implement custom page management policies. In contrast, the soft-realtime requirements of continuous media motivated self-paging in Nemesis [Han99]. In a self-paging system, faults are redirected to the application itself, to ensure resource accountability. As with AVM, the target hardware is a uniprocessor with a software-loaded TLB. A similar upcall mechanism for reflecting page faults was used in K42 [KAR+06]. This style of fault reflection, apart from allowing the system to account fault handling time to the application which caused the fault, also enables applications to implement custom page replacement and fault handling policies.

In contrast, extensible kernels like SPIN [BSP+95] and VINO [ESG+94] allow downloading of safe policy extensions into the kernel for performance. For example, SPIN's kernel interface to memory has some similarity with Barrelfish's memory system's user-space API: `PhysAddr` allowed allocation, deallocation, and reclamation of physical memory, `VirtAddr` managed a virtual address space, and `Translation` allowed the installation of mappings between the two, as well as event handlers to be installed for faults. In comparison, Barrelfish's memory system allows applications to define policies completely in user-space, whereas SPIN has to rely on compiler support to make sure the extensions are safe for use in kernel-space.

Dune

More recently, Dune [BBM+12] has shown how Linux can be extended to utilize virtualization hardware in modern processors to provide a process, instead of a machine abstraction. This enables applications to utilize processor features that have previously been unavailable to applications such as ring protection, page tables, and tagged TLBs. In context of this dissertation, this enables Dune applications to autonomously build their address space by directly programming the MMU. Dune provides protection by controlling the guest-to-host mappings which control what "physical" addresses mean for the application which is running in the same environment as a virtual machine guest OS kernel.

Mach

The goal of Mach [RTY+88] is to provide a portable multiprocessor operating system. One of the goals of the project is to explore the relationship between hardware and software memory architectures and to design a memory management system that would be readily portable to both multiprocessor and uniprocessor machines. Mach's memory system supports five core features: large and sparse virtual address spaces, copy-on-write virtual memory operations, copy-on-write and read-write memory sharing between tasks, memory-mapped files, and user-provided backing store objects and pagers. While the first four features, which at the time were not available in UNIX, have found their way into modern Linux, Mach's user-provided backing store objects and pagers remain fairly unique, and I will discuss this feature in more detail here as its influence can be clearly seen in Barrelfish's memory system which I describe in chapter 3.

Notably Mach achieves all these "modern" memory system features without making its internal memory representation depend on any specific architecture. Rather the opposite is true: Mach makes relatively few assumptions about available memory management hardware. The primary hardware feature which is required by Mach is the ability to handle and recover from page faults. Mach doesn't make any restrictions on the system page size other than requiring it to be a power of two multiple of the hardware page size and allows setting it as a boot parameter.

Mach has five basic abstractions: tasks, threads, ports, messages and memory objects. In Mach a *task* is the execution environment in which threads may run. A task includes a paged virtual address space and protected access to system resources. A task's virtual address space consists of an ordered collection of mappings to memory objects. A *thread* is the basic unit of CPU utilization. All threads within a task share all the resources which are allocated to that task. A *port* is a communication channel, i.e. a queue for messages protected by the kernel. A *message* is a typed collection of data objects used in communication between threads. Messages may be of any size, and may contain pointers and typed capabilities for ports. A Mach *memory object* is a collection of data which is provided and managed by a server and which can be mapped into the virtual address space of a task.

As any operations on objects other than messages have to be performed by sending messages to ports, Mach permits system services and resources to be managed by user-space tasks. In fact, the Mach kernel itself can be considered as a task with multiple threads. The kernel task acts a server which in turn implements tasks, threads and memory objects. Creating a task, thread or memory object returns access rights to a port which represents the new object and can be used to manipulate it.

Basic VM operations

Each mach task has a large address space which is made up of mappings between ranges of memory addressable to the task and memory objects. The size of a virtual address space is only limited by the restrictions of the underlying hardware. A task can modify its address space in different ways, which include allocating a regions of virtual memory on a page boundary, deallocating a region of virtual memory, setting the protection status of a virtual region, specifying the inheritance of a virtual region, and creating and managing a memory object which can then be mapped into the address space of another task.

Mach allows both copy-on-write and read-write sharing of memory between tasks. A virtual page's inheritance value controls sharing between the task and its children. The inheritance can be set to shared, copy, or none. Pages marked as shared are shared for read and write. Pages marked as copy are logically copied to the child, but internally copy-on-write is employed for efficiency. Pages marked as none are not available in the child, and the corresponding virtual region is not allocated. Similarly, protection is specified on a per-page basis. Each page has two protection values: the current and the maximum protection. The maximum protection specifies the maximum permissions that the page may have, and the current protection controls the actual hardware permissions. The maximum protection cannot be raised, but it can be lowered. If the maximum protection is lowered below the current protection, the current protection is set to the new maximum protection. Each protection is a combination of read, write and execute permissions. Enforcement of the protection depends on hardware support.

Mach virtual memory

Mach's virtual memory implementation uses four different data structures: (1) the resident page table, which is used to keep track of information about machine-independent pages, (2) the address map, which is a doubly linked list of entries, each of which describes a mapping from a range of addresses to a region of a memory object, (3) the memory object, which is a unit of backing storage managed by the kernel or a user task, and (4) the pmap, which is a machine dependent memory mapping data structure, e.g. the hardware defined page tables.

Mach maintains all the important virtual memory information in machine independent code. The machine dependent part only maintains those mappings which are essential for system operation, such as the kernel map and frequently referenced task addresses, and is allowed to garbage collect non-important mapping information. Notably the machine dependent part is not required to maintain full knowledge of valid mappings from virtual addresses to hardware pages.

The address map keeps track of mappings from contiguous ranges of virtual addresses onto contiguous areas of memory objects. Each address map entry must map to a contiguous area of a single memory object.

Memory objects Address maps do not have to keep track of backing storage, because Mach implements all backing store as memory objects. A memory object is a repository for data, indexed by byte, on which operations such as read and write can be performed. In many respects memory objects are similar to UNIX files.

Each memory object is reference counted. This allows memory objects to be garbage collected when all mapped references to them are removed. To speed up periodical reuse of memory objects, e.g. text sections or other

frequently used files, Mach maintains a cache of frequently used memory objects which will not be garbage collected when their reference count hits zero. Any pager may use domain specific knowledge to request that an object is kept in this cache after it is no longer referenced.

Each memory object is associated with a managing task, which is called a *pager*. This association enables the ability to handle page faults and page-out requests outside the kernel. Access to the pager is represented by a port to which the kernel can send messages to request data or to notify the pager about a change in the object's primary memory cache. In addition to the pager port, the kernel maintains a unique identifier for each memory object, which is also represented by a port. The kernel manages the pages which are currently cached in primary memory through the kernel paging daemon. All other pages are stored and fetched by the pager. The pager has another port which it can use to send messages to the kernel to manage the object or its primary page cache.

A pager may be internal to the Mach kernel, or an external user-state task. Memory which has no associated pager is automatically zero filled and page-out is done to a default pager which is provided by the kernel.

Physical address maps The physical address map (pmap) is Mach's machine dependent memory management code. Its purpose is to manage the translation hardware. The pmap implementation is has to provide page level operations on the translation hardware data structures and has to ensure that the appropriate hardware translation is operational whenever the state of the machine needs to change from kernel to user state or vice versa.

An overview of Barrelfish

In this section we provide an overview of Barrelfish. Barrelfish is the original implementation of the multikernel OS architecture [BBD+09]. In the multikernel model, the OS is structured as a distributed system of cores that communicate using messages and share no memory. We show a stylized figure depicting the multikernel model in figure 2.4. The multikernel model is guided by three design principles:

1. Make all inter-core communication explicit.
2. Make OS structure hardware-neutral.
3. View state as replicated instead of shared.

These principles allow the OS to gain improved performance by reusing algorithms designed for distributed systems, seamlessly enables the OS to support heterogeneous hardware, and improves modularity.

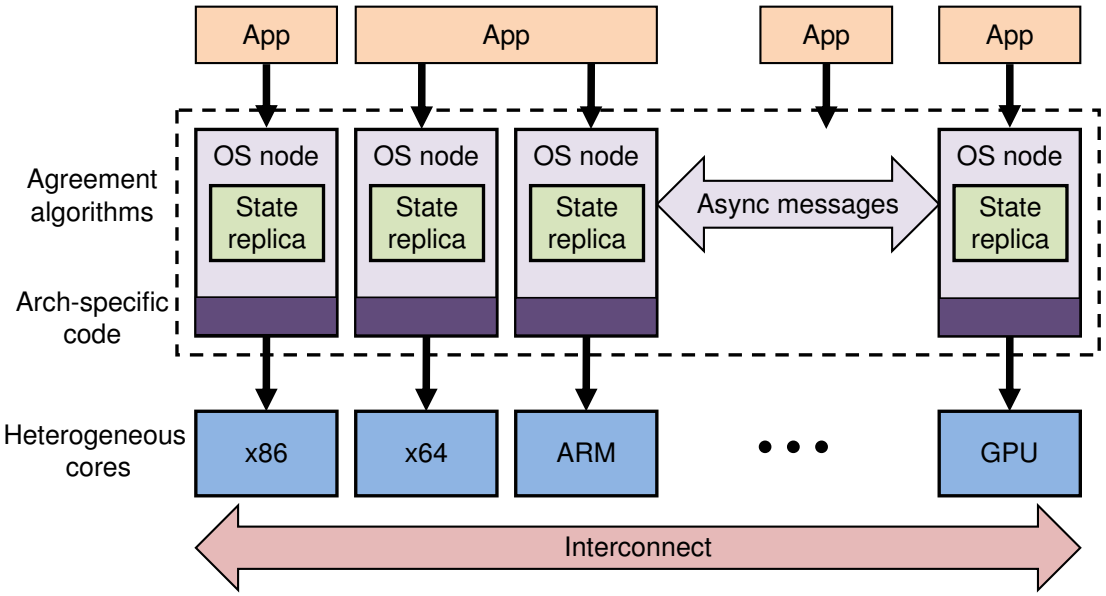


Figure 2.4: *The multikernel model*

In the rest of this section, we discuss specific parts of Barrelfish that are relevant in the context of this thesis.

Domain specific languages

Barrelfish makes liberal use of domain specific languages (DSL) to describe various parts of the system. There are multiple reasons why. First, with careful use of DSLs, it becomes possible, if not easy, to formally reason about parts of the system. Second, DSLs eliminate the need to write repetitive bits of code which are easy to get wrong such as accessing bits in device registers, and argument marshalling for IPC. Third, as a consequence of the second point, DSLs it easier to add new IPC interfaces etc, as the amount of tedious glue code that needs to be written manually is decreased significantly.

The most prominent domain specific languages in Barrelfish deal with inter-process messages, and access to device registers respectively. However, there are a number of other DSLs, for purposes such as describing error codes in a system-wide unique way, defining trace point types for Aquarium, the Barrelfish trace analysis framework [SG13], and more recently describing relationships between address spaces etc. in Sockeye [Sch17].

Naturally, given the prevalence of domain specific languages in Barrelfish, we also use a DSL called Hamlet to define and describe the set of Barrelfish capability types [DBR09]. While the DSL currently only deals with the types themselves, and their relationships with each other, there are long-standing plans to also describe the possible invocations on each type in the DSL and alleviate the need of writing argument (un)marshalling code by hand when implementing new capability types and their associated invocations.

Capabilities in Barrelfish

Type system

Barrelfish capabilities are strongly typed. This idea was adapted from seL4, which had a relatively small number of capability types which express all possibilities in which new capabilities can be derived from existing ones. Traditionally, capability types are used to specify, at runtime, the purpose of regions of physical memory. In a simple partitioned capability system, there might be only two types: either memory is used for capability storage, or available for general usage. In such a system, the operating system would then ensure that a region that is reserved for capability storage will never be directly accessible from a user application.

As mentioned in section 2.6.1, Barrelfish defines its capability types in a DSL called Hamlet.

Retype

With a capability type system, we need a new operation on capabilities, *retype*. Retype is the capability operation that enables users to create new capabilities from existing ones which they hold. The retype operation will fail if the requested capability does not match the constraints given the source capability and the static type system that is defined in the operating system.

Capability types in Barrelfish

Barrelfish, in May 2018, has 50 distinct capability types. In Barrelfish, every capability that refers to an addressable region on the memory bus is derived from the type `PhysAddr` which simply identifies a range of addresses

on the memory bus. The first distinction is then made between memory-mapped device registers (`DevFrame`) and general purpose memory (`RAM`). From `RAM`, all the semantic capability types are then derived. The capability storage region type in Barrelfish is called `CNode`, after seL4. RAM that is mappable by user applications is retyped to `Frame`. In contrast to seL4 however, Barrelfish distinguishes between data regions and page tables and has a unique type for each type of page table for each architecture. We discuss the importance and usefulness of having explicit page table types in chapter 3 where we present our inverted memory system. There are a few other specially-typed memory regions which are available in Barrelfish. The kernel control block (`KCB`) is a special region which contains all pointers that might be global variables in another kernel [ZGKR14]. The dispatcher control block (`Dispatcher`) type is used for per-process metadata which has to be shared between the process and the kernel.

For each capability type that can be mapped into an application's address space there exists a companion `Mapping` type. Every time a capability is mapped into an application's address space, a mapping capability of matching type is created to track that particular mapping throughout its lifetime. Mapping capabilities and alternatives are presented in chapter 3.

Barrelfish also has quite a few capability types that do not refer to regions of physical addresses, but rather convey authority over other system resources, such as unique identifiers (`IDCap`), interrupt sources and destinations (`IRQSrc` and `IRQDest`), performance monitoring access, IPI access, and privileged kernel interface access.

A Barrelfish application's view of capabilities

In this section we discuss how Barrelfish stores the capabilities for a single application and how the application can use its capabilities.

Applications refer to capabilities by *capability address*. A capability address is a 32 bit integer. This allows an application to refer to 2^{32} distinct capabilities.

The capabilities of an application are stored in its capability address space (*CSpace*), which is a two-level table stored in regions of memory which the application cannot map into its virtual address space. Barrelfish enforces that each second level table is a 16kB memory region. As our capabilities take up 64 bytes of storage, each such second-level table can hold 256 capabilities. Therefore the least significant eight bits of the capability address provide the index into a second-level table. The remaining twenty-four bits represent the index of the slot in the first-level table that is holding the capability for the second-level table.

Barrelfish uses two distinct capability types for the first and second level tables: `L1CNode` and `L2CNode`. However, throughout this dissertation, we often use *CNode* to refer to both types, as the two specific types are mainly an implementation detail and not required for the high-level design.

Capability addresses and `caprefs`

Even though the kernel only understands capability addresses, the standard Barrelfish library OS encapsulates the capability addresses in capability references (*capref*) which are just an alternate representation of the capability's location in the application's CSpace.

A big reason for having a second representation of capability addresses is to have a layer of abstraction between application code – which uses caprefs – and the specifics of the layout and construction of the CSpace. This allowed us to completely restructure the CSpace layout from generic guarded page tables to the two-level layout discussed above without having

to touch most of the application code that exists in Barrelfish by keeping the capref structure unchanged.

A capref is made up of a *cnoderef* and a slot number. The slot number corresponds to the index of the slot holding the capability in the second-level table.

```
struct capref {
    struct cnoderef cnode;
    cslot_t slot;
};
```

To allow `capref`s to refer to both L1 and L2 CNode slots in the local capability space, the `cnoderef` contains enough information to distinguish L1 and L2 caprefs without needing separate struct definitions.

```
struct cnoderef {
    capaddr_t croot;
    capaddr_t cnode;
    enum cnode_type level;
};

enum cnode_type {
    CNODE_TYPE_ROOT = 0,
    CNODE_TYPE_OTHER,
    CNODE_TYPE_COUNT,
};
```

For `capref`s referring to slots in the application's CSpace the `croot` field is always 0x2. This is the capability address of the application's root CNode in its own CSpace.

The `level` field in a `capref`'s `cnoderef` indicates whether a `capref` refers to a slot in the root (L1) CNode or a slot in a L2 CNode capability.

The `cnode` field of `capref`s referring to slots in the application's root (L1) CNode is ignored, and the value of the `capref`s `slot` field is shifted left by

eight bits (remember: L2 CNodes resolve the last eight bits of a capability address), to create a capability address that can be used by the CPU driver to find the L2 CNode. The CPU driver also requires the `level` information to be able to stop address resolution at the appropriate point. This is required because L2 CNode capabilities are stored in the memory region that makes up the L1 CNode capability.

For capability slots in L2 CNodes, the `cnode` field of the cnoderef is the index in the L1 CNode in which the L2 CNode is stored, shifted left by eight bits. This allows easy capability address constructions for L2 slots, by just performing a bitwise OR on the `slot` field stored in the `capref` and the `cnode` field stored in the capref's cnoderef.

Initial CSpace Layout

All Barrelfish applications start with a well-defined initial CSpace layout. Given the assumption that most applications do not need more than approximately sixty-thousand capabilities at a single point in time, we construct initial CSpaces with an L1 CNode with 256 slots. In the implementation, we often use the term *root CNode* to refer to an application's L1 CNode.

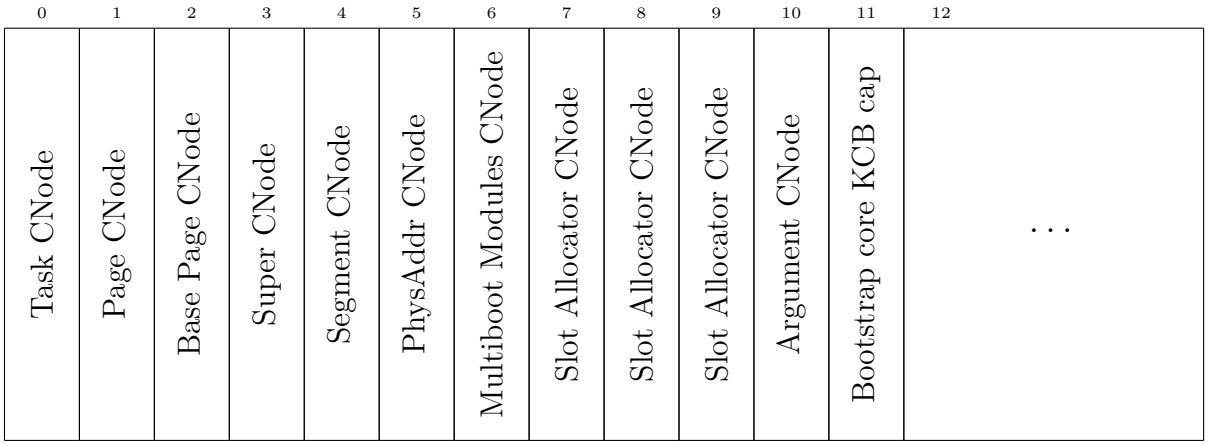


Figure 2.5: *Well-defined root CNode slots*

Root CNode The root CNode is the top-level CNode of the application's CSpace and has to be of type `L1CNode`. The root CNode is the entry point for any capability lookup initiated by the domain. Figure 2.5 lists the well-defined slots in the root CNode.

Task CNode The task CNode holds capabilities which the library OS requires throughout the application's life time. The well-defined slots of the task CNode are shown in figure 2.6. Some of the task CNode's slots may be empty for most domains, e.g. only the monitor's get a copy of the Kernel capability.

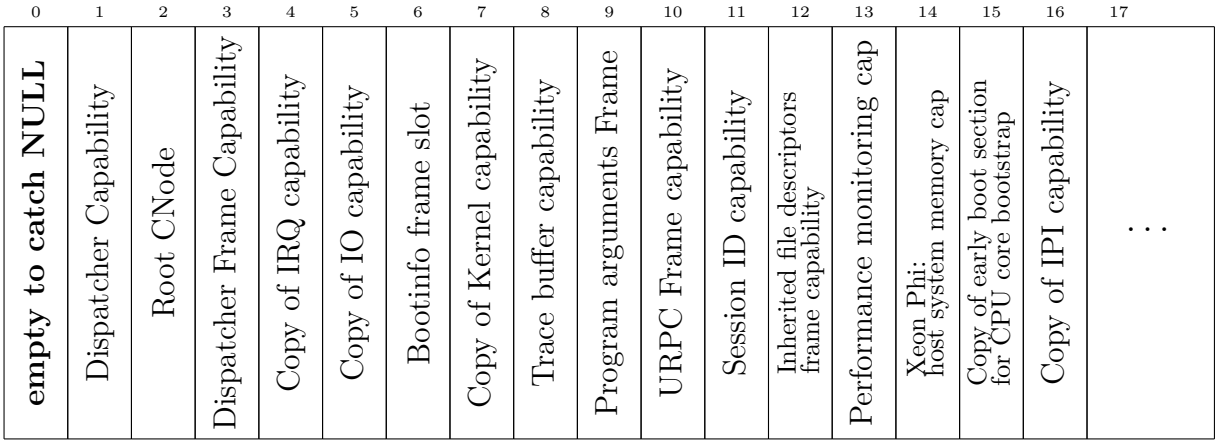


Figure 2.6: *Well-defined task CNode slots*

Page CNode The first slot in the page CNode is always the application's root page table. Further slots in the page CNode are used to store capabilities to the application's initial page tables and mapping capabilities that are created before the application runs.

We will call the application which builds an application's initial CSpace and address space the *spawning application*.

Base Page CNode The base page CNode is filled with 256 4kB RAM capabilities by the spawning application. These RAM capabilities can be used by the application for early allocations before it has established a connection to the memory server.

Super CNode For the memory server, the super CNode is filled with all the RAM capabilities that it will manage.

Segment CNode The segment CNode contains with the Frame capabilities which store the application's runtime segments, such as `.text`, `.data` and `.bss`.

PhysAddr CNode The PhysAddr CNode contains platform-specific data such as ACPI tables. This CNode only exists for applications that require these regions.

Multiboot Modules CNode For `spawnd`, which is the Barrelfish process that has the authority to create new dispatchers[Dam17], the multiboot modules CNode exists, and contains all the multiboot module Frame capabilities. The multiboot module Frame capabilities refer to the regions of RAM that contain the multiboot module data.

Slot Allocator CNodes Each application gets three empty L2 CNodes that are used to initialize the default capability slot allocators in the library OS.

Argument CNode The argument CNode is created by `spawnd` and can be used to pass a new application arbitrary capabilities. There is no

predefined behaviour for capabilities in this CNode, but the spawning application and the spawned application have to agree on the meaning of each entry.

Slot allocation

Barrelfish provides library code which allows applications to allocate new capability slots. There are a few different slot allocators which can be used depending on the application's needs. A lot of library code uses the *default* slot allocator which is composed from a number of *single* slot allocators. A single slot allocator manages the slots in a single CNode.

The *default* slot allocator, which manages slots without special requirements is composed of a single slot allocator for slots in the root CNode and a single slot allocator for each L2 CNode that has been allocated over time. Initially, the default slot allocator is seeded with the CNodes located in slots 8 and 9 of the root CNode. The default slot allocator is an instance of the *twolevel* slot allocator, which can also be instantiated for another application's CSpace.

Resizing the root CNode

Even though we split the capability addresses into only two levels of CNode lookup, we do not want to fully allocate a root CNode with 2^{24} slots when we create a domain. We instead choose to create root CNodes with 256 slots initially. To address the needs of domains that require many capability slots, i.e. more than approx. $(256 - 8) \times 256 = 63488$, we need a way to dynamically resize the root CNode. We implement CNode resizing for L1 CNodes by introducing a new invocation on the L1 CNode. This invocation takes two capability addresses as arguments. The first capability address

has to point to a RAM capability which will be transformed into the new root CNode. The second capability has to point to an empty slot where the CPU driver will store the old root CNode capability which the domain can then safely delete. It is important to note that this invocation needs to switch out the domain's CNode atomically from the point of view of the application.

Referring to other CSpaces

There are situations in which a domain will have to access a CSpace other than its own. This is most prevalent in the monitor and when creating the initial CSpace for a new domain. To support these use cases, we change the invocations that take capability addresses as arguments to take an extra argument specifying the L1 CNode that should be used to perform the capability lookup. The exception is the capability on which the invocation is called, the *target* capability. The target capability always has to exist in the calling domain's CSpace. Where necessary the default library OS (`libbarrelfish`) transparently creates invokable copies of capabilities stored in a CSpace that is not the domain's.

Using capabilities

In many ways, Barrelfish capabilities are similar to Unix file descriptors which carry extra information, such as a rich type which identifies the purpose of the object which the capability refers to.

Barrelfish applications often use capabilities in ways that are not visible to the application programmer, when the programmer uses features which are provided by the default Barrelfish library OS. Many of the features provided by the default library OS are reminiscent of features provided by a typical Unix-style kernel.

However, applications that care about memory often will not simply use malloc and free but explicitly request capabilities to physical memory from a system service called the memory server which is responsible for managing all the RAM in a machine.

Apart from a selection of capabilities that an application is provided in its initial CSpace as discussed above, it can get capabilities of type RAM from the memory server.

Applications can use the retype operation on parts (or all) of a RAM capability to create specific page table capabilities, or capabilities for data pages. Whenever an application requests a retype from the Barrelfish CPU driver it will have to provide a sufficient number of contiguous destination capability slots for the CPU driver to store the newly created capabilities.

Because applications can receive capabilities over a message channel, each Barrelfish message channel always holds the capability address of an empty slot in the receiving side's CSpace which the CPU driver can use to store a received capability in the receiver's CSpace.

An application is free to create copies of any capabilities in its CSpace. Similarly, an application can at any point delete any capability in its CSpace. Finally, an application can *revoke* capabilities in its CSpace. The revoke operation can be thought of as a series of deletes. Revoke deletes any copies and descendants of the capability on which the operation is executed. Descendants in this context are capabilities which were created from the original capability using the retype operation.

Message passing

According to the first design principle of the multikernel model, "Make all inter-core communication explicit", Barrelfish applications communicate

using explicit message channels by default. The message passing subsystem is fairly sophisticated and allows applications to define both asynchronous and remote procedure call message channels. Accordingly, a native Barrelfish application is usually designed to be event-driven, as all communication with operating system services and other native applications will use messages. This design principle however, does not preclude applications from creating shared memory regions. Notably, most existing Barrelfish applications that utilize shared memory regions use the message passing system to bootstrap their shared memory regions. The message passing system is an attractive bootstrap medium for shared memory as it allows applications to send copies of capabilities for which they have sufficient permissions to other applications.

User-space memory management

User-space memory management is another prominent feature of Barrelfish and is one of the contributions of this dissertation. We explore and discuss the design and implementation of the memory system in the next chapter of this dissertation. In a nutshell, Barrelfish leverages its capability system to give applications control over their virtual address space in a controlled manner. This allows applications to tailor their address space to their specific requirements, such as NUMA placement, mapping granularity, etc. Many aspects of Barrelfish's memory system, such as virtual regions, memory objects and physical maps, are reminiscent of Mach's virtual memory as discussed in section 2.5.4.

3

Design and implementation on a single core

Barrelfish adopts a radically inverted view of memory management compared with classical demand-paged VM. Barrelfish processes run in a virtual address space (the MMU is enabled) but this address space is constructed by the application itself, and may vary across cores within the process.

Our key motivation to take this approach to virtual address spaces for applications is this: the performance, scale, and heterogeneity of hardware platforms means that application- or workload-specific optimizations of virtual memory are critical for performance, and this performance is now a "hard" requirement for many applications.

While Barrelfish allows great flexibility in arranging an address space, it nevertheless ensures the following key safety property.

Invariant 3.1. *No Barrelfish process can issue read or write instructions for any area of physical memory for which it does not have explicit access rights.*

Subject to this requirement, Barrelfish also provides the following completeness property.

Invariant 3.2. *A Barrelfish process can create any address space layout permitted by the MMU for which it has sufficient resources.*

In other words, Barrelfish itself poses no restriction on how the memory hardware can be used.

There are three main challenges in the implementation that Barrelfish's memory system must address: Firstly, it must securely name and authorize access to, and control over, regions of physical memory. In Barrelfish, we leverage the capability system presented in chapters 2 and 4 of this dissertation to address the first challenge. Secondly, it must allow safe control of hardware data structures (such as page tables) by application programs. This, is achieved by considerably extending the set of memory types supported by the capability system in Barrelfish (compared to seL4) to capture memory-management specific meaning in the capability types. Finally, Barrelfish's memory system must give applications direct access to information provided by the MMU (such as access and write-tracking bits in the page tables). Unlike prior approaches which rely on virtualization technology, Barrelfish's memory system allows direct read-only access to page table entries; we explain below why this is safe.

Barrelfish's memory system has three main components: First, the kernel provides capability invocations that allow application processes to install, modify and remove page table entries and query for the base address and size of physical regions. Second, the kernel exception handler redirects any exceptions generated by the MMU to the application process that caused the exception. Thirdly, a runtime library provides to applications an abstraction layer over the capability system which exposes a simple, but expressive API for managing page tables.

Physical memory allocation

Barrelfish applications directly allocate regions of physical memory and pass around authorization for these regions in the form of capabilities. Regions can be mapped into a virtual address space by changing a page table, or used for other purposes such as holding page tables themselves.

For the purpose of expressing constraints on address space construction, we extend the Barrelfish capability design, itself inspired by seL4 [EDE08, KEH+09, DEE06].

In seL4, all physical regions are represented by capabilities, which also confer a particular *memory type*. As discussed earlier, the integrity of the capability system itself is ensured by storing capability representations in memory regions of type `CNode`, which can never be directly written by user-space programs. Instead, a region must be of type `Frame` to be mapped writable into a virtual address space. Holding both `Frame` and `CNode` capabilities to the same region would enable a process to forge new capabilities by directly manipulating their bit representations, and so is forbidden. Such a situation is prevented by having the kernel enforce a type hierarchy for capabilities.

Capabilities to memory regions can be split and *retyped* according to a set of rules. At system start-up, all memory is initially of type `Untyped`, and physical memory is allocated to processes by splitting the initial untyped region. Retyping and other operations on capabilities is performed by system calls to the kernel.

seL4 capabilities are motivated by the desire to prove correctness properties of the seL4 kernel, in particular, the property that no system call can fail due to lack of memory. Hence, seL4 and Barrelfish perform no dynamic memory allocation in the kernel, instead memory for all dynamic kernel data

structures is allocated by user-space programs and retyped appropriately, such as to a kernel thread control block or a `CNode`, for example.

In the context of providing a rich memory system API, capabilities are attractive since they export physical memory to applications in a safe manner: application may not arbitrarily use physical memory; they must instead "own" the corresponding capability. Furthermore, capabilities can be passed between applications. Finally, capabilities have some characteristics of objects: each capability type has a set of *operations* which can be invoked on it by a system call. We call these operations *invocations*.

In Barrelfish, and seL4, the kernel enforces safety using two types of metadata: a *derivation database*, the Barrelfish implementation of which we discuss at length in chapter 4 and a per-processes *capability space*. We have presented a more detailed discussion of Barrelfish's capability space in section 2.6. All capability objects managed by a kernel are organized in a capability derivation tree. This tree enables efficient queries for descendants (of retype and split operations) and copies. These queries are used to prevent retype races on separate copies of a capability that might compromise the system.

User processes refer to capabilities and invoke operations on them using opaque handles, so-called capability references, as presented in section 2.6.3. Each process has its own capability address space, which is explicitly maintained via a two-level tree in the kernel which functions similar to a regular two-level page table, but with a dynamically-sized root level table. The nodes of the tree are also capabilities (retyped from RAM capabilities) and are allocated by the application.

The root of the capability tree for each process is stored in the process control block. When a process invokes a capability operation it passes to the kernel the capability handle with the invocation arguments. To perform

the operation, the kernel traverses the process' capability space to locate the capability corresponding to the handle and authorizes the invocation.

For the memory system we build on the basic Barrelfish capability mechanisms to allow explicit allocation of different kinds of memory. A memory region has architectural attributes such as the memory controller it resides on, whether it is on an external co-processor like a GPGPU or Intel Xeon Phi, whether it is persistent, etc. Applications explicitly acquire memory with particular attributes by requesting a capability from an appropriate memory allocator process, of which there may be many. Furthermore, less explicit "best effort" policies can be layered on top by implementing further virtual allocators which can, for example, steal RAM from nearby controllers if local memory is scarce.

Securely building page tables

Page tables are hardware specific, and at the lowest level, Barrelfish's interface (like seL4) reflects the actual hardware. Applications may use this interface directly, or a high-level API with common abstractions for different MMUs, to safely build page tables, exchange page tables on a core, and install mappings for any physical memory regions for which the application is authorized. The choice of virtual memory layout, and its representation in page tables, is fully controlled by the application. Cores can share sub-page-tables between different page-table hierarchies to alias a region of memory at a different address or to share memory between different cores as in Corey [BWCC+08].

The work done for this dissertation adds support for multiple page sizes (2 MB and 1 GB superpages in `x86_64`, and 16 MB, 1 MB, and 64 kB pages in ARMv7-a [ARM]) to the original Barrelfish memory management sys-

tem [BBD+09]. Barrelfish's memory system decouples the physical memory allocation from programming the MMU. Therefore the API allows for a clean way to explicitly select the page size for individual mappings, map pages from a mixture of different page sizes, and change the virtual page sizes for mappings of contiguous physical memory regions all directly from the applications itself instead of relying on the kernel to implement the correct policy for all cases.

To do this, Barrelfish's memory system seL4's set of capability types by introducing a new capability type for every level of page table for every architecture supported by the OS. This is facilitated by the *Hamlet* domain-specific language for specifying capability types [DBR09].

For example, for an MMU in `x86_64` long-mode there are four different types of page table capability, corresponding to the 4 levels of a 64-bit x86 page table (`PML4`, `PDPT`, `PD`, and `PT`). A `PT` (last-level page table) capability can only refer to a 4k page-aligned region of RAM and has a `map` operation which takes an additional capability plus an entry number as arguments. This capability in turn must be of type `Frame` and refer to another 4k page. The operation installs the appropriate page table entry in the `PT` to map the specified frame. The kernel imposes no policy on this mapping, other than restricting the type and size of capabilities.

Similarly, a `map` on a `PD` (a 2nd-level "page directory") capability only accepts a capability argument which is of size 4 kB and type `PT`, *or* of type `Frame` and size 2 MB (signifying a large page mapping).

A small set of rules therefore captures all possible valid and authorized page table operations for a process, while excluding any that would violate the safety property. Moreover, checking these rules is fast and is partly responsible for Barrelfish's memory system's superior performance described in section 3.6.2. This type system allows user-space Barrelfish programs to

construct flexible page tables while enforcing the safety property stated at the start of this section.

Barrelfish's full kernel interface contains the following capability invocations on page table types: `identify`, `map`, `unmap`, `modify_flags` (protect), and `clear_dirty_bits`.

Memory regions represented by capabilities and associated rights allow user-level applications to safely construct page tables; they allocate physical memory regions and retype them to hold a page table and install the entries as needed.

Typed capabilities ensure a process cannot successfully map a physical region for which it does not have authorization. The process of mapping itself is still a privileged operation handled by the kernel, but the kernel must only validate the references and capability types before installing the mapping. Safety is guaranteed based on the type system: page tables have a specific type which cannot be mapped writable.

Care must be taken in Barrelfish to handle capability revocation. In particular, when a `Frame` capability, or any other mappable capability, is revoked, all page table entries for that frame must be quickly identified and removed. Barrelfish handles this by creating mapping capabilities whenever a mapping is installed. The caller must supply an empty capability slot in every map invocation. The kernel stores the newly created mapping capability in that slot. These mapping capabilities can then be used to (i) manage the mapping which they refer to and (ii) give the kernel a way to efficiently find all the mappings for a given `Frame` capability. We will discuss mapping capabilities and alternate strategies of keeping track of mappings in more detail in section 3.3 of this dissertation.

As described so far, each operation requires a separate system call per page table entry we wish to modify. Barrelfish optimizes this in a straightforward

way by allowing batching of requests, amortizing system call cost for large region operations. The `map`, `unmap`, and `modify_flags` operations all take multiple consecutive entries for a given page table as arguments.

In section 3.6.3 we confirm existing work on the effect of page size on performance of particular workloads, and in section 3.6.4 we show that the choice of the page size is highly dynamic and depends on the program's configuration such as the number of threads and where memory is allocated.

In contrast, having the OS transparently select a page size is an old idea [NIDC02] and is the default in many Linux distributions today, but finding a policy that satisfies a diverse set of different workloads is difficult in practice and leads to inherent complexity with questionable performance benefits [GLD+14, GH12, Cas13, San].

Keeping track of virtual to physical mappings

As the basic method of authorization in Barrelfish is a capability, it is only natural that we would like to keep track of virtual to physical mappings using capabilities. The most natural way of doing so would be to make the page tables be CNodes, and the act of copying a Frame (or other mappable) capability into a page table CNode would install the corresponding virtual to physical mapping. However, while this works great on systems without a hardware page table walker, the presence of a hardware page table walker usually defines a page table format that is not compatible with 64 byte capabilities. Another downside of the CNode approach is that mapping parts of a Frame capability, or mapping Frame capabilities larger than the base translation granularity is non-trivial, and can have considerable overhead in the number of capabilities required. Therefore we need to come

up with a method of keeping track of how page table entries and Frame capabilities correspond to each other.

Throughout the research for this dissertation we have considered a number of methods to keep track of this correspondence. The first method we discuss is keeping shadow page tables alongside the page tables that have the correct format for the hardware page table walker. This method, however, suffers from the same drawbacks as discussed previously for the "CNode" style approach, namely, there is a proliferation of capabilities when creating large mappings, and the memory overhead for mostly empty page tables is not insignificant, as we would reserve an extra 32 kB per 4 kB `x86_64` hardware page table, equating to a memory overhead of 8x.

The second approach is to store alongside each Frame capability where it is mapped. This approach has the problem that now each copy of a capability can be mapped at most once, which is a shift in semantics from the capability model discussed earlier where we say that if you hold a copy of a capability you can map the memory it refers to arbitrarily often. Apart from that semantic shift, this approach also leads to situations where an application has to create many copies of a capability just for the purpose of creating the mappings it requires. Additionally, in the prototype implementation for this approach, we chose to store the mapping reference outside the type-specific part of the capability representation, which produces a system where a lot of capabilities have up to 24 bytes of metadata that is never used, and in fact cannot be used because there are a number of capability types that cannot be mapped into an application's virtual address space. This approach has been discussed in depth in my master's thesis [Ger12].

The final option we consider is to create new capability types, *Mapping* capabilities, which are created whenever a mapping is inserted into a page table. Mapping capabilities appear as descendants of the mapped capability in the capability derivation tree and store both a pointer to the capability

that is mapped, the page table in which the Frame is mapped, the first entry in the page table which belongs to the mapping, and the number of entries the mapping occupies in the page table. Before settling on these elements, the mapping capability used to contain the offset into the mapped capability, but that information can be computed from the first page table entry of the mapping and the base address of the mapped capability. This method restores the semantics of being able to map the same capability multiple times, and only consumes space proportional to the number of mappings rather than the number of allocated page table entries or number of capabilities in the system.

Barrelfish currently uses Mapping capabilities to keep track of mappings. The implementation provides one distinct mapping capability type for each type that is mappable. Each mapping capability is situated as a child of the corresponding mappable type in the derivation tree. However, mapping capabilities cannot be created by calling retype on the mappable type explicitly. Rather, the caller needs to supply the map invocation with an extra empty capability slot which gets populated with the newly created mapping capability during the map invocation. Once created, a mapping can be modified and removed by using invocations on the returned mapping capability. Additionally, we still allow mappings to be removed by using an invocation on the page table containing the mapping.

Conversely, if a user deletes, either directly or through a revoke, a mapped capability, the CPU driver finds all mapping capabilities that are descended from that particular capability and clears the page table entries that were used for each mapping. This ensures that the user cannot circumvent the capability system by using stale page table entries for regions of memory to which they do not have authorization anymore.

Page faults and access to status bits

The memory system uses the existing Barrelfish functionality for reflecting VM-related processor exceptions back to the faulting process, as in Nemesis [Han99] and K42 [KAR+06]. This incurs lower kernel overhead than classical VM and allows the application to implement its own paging policies. In sections 3.6.1 and 3.6.2 we show that Barrelfish's trap latency to user space is considerably lower than in Linux.

We extend Barrelfish to allow page-traps to be eliminated for some use-cases when the MMU maintains page access information in the page table entries. While Dune [BBM+12] uses nested paging hardware to present "dirty" and "accessed" bits in an `x86_64` page table to a user space program, Barrelfish's memory system achieves this *without* hardware support for virtualization.

We extend the kernel's mapping rules in section 3.2 to allow page tables themselves to be mapped read-only into a process' address space. Essentially, this boils down to allowing a 4 kB capability of type `PML4`, `PDPT`, `PD`, or `PT` to be mapped in an entry in a `PT` instead of a `Frame` capability, with the added restriction that the mapping must be read-only.

This allows applications (or libraries) to read "dirty" and "accessed" bits[1] directly from page table entries without trapping to the kernel. Setting or clearing these bits remains a privileged operation which can only be performed by a kernel invocation passing the capability for the page table.

Note that this functionality remains safe under the capability system: an application can only access the mappings it has installed itself (or for which it holds a valid capability), and cannot subvert them.

In section 3.6.5 we demonstrate the benefits of this approach for a garbage collector.

[1] for architectures which have those bits in the hardware page tables

Since Barrelfish's memory system doesn't need hardware virtualization support, such hardware, if present, can be used for virtualization. Barrelfish's memory system can work both inside a virtual machine, or as a better memory management system for a low-level hypervisor.

Moreover, nested paging has a performance cost for large working sets, since TLB misses can be twice as expensive. In section 3.6.6 we show that for small working sets (below 16 MB for our hardware) a Dune-like approach outperforms Barrelfish due to lower overhead in clearing page table bits, but for medium-to-large working sets Barrelfish's lower TLB miss latency improves performance.

The Barrelfish and Dune approaches are complementary, and a natural extension to Barrelfish, which was not explored in this dissertation, would allow applications access to both the physical (machine) page tables *and* nested page tables if the workload can exploit them.

High-level convenience

Barrelfish provides a number of APIs above the capability invocations discussed above. In this section we will look at a number of these higher-level APIs which are provided by the default Barrelfish library OS and are implemented using the primitives that make up Barrelfish's memory system.

User space virtual address space management

Barrelfish's library OS provides an API which is reminiscent of Mach's memory system and keeps track of the application's virtual address space layout.

While applications can directly use the invocations that make up Barrelfish's memory system, the library OS builds on top of the memory system to provide higher-level abstractions based on the concepts of *virtual regions* (contiguous sets of virtual addresses), and *memory objects* that can be used to back one or more virtual regions and can themselves be comprised of one or more physical regions.

This abstraction is important to improve the usability of Barrelfish's memory system.

Manually invoking operations on capabilities to manage the virtual address space can be cumbersome; take the example of a common operation such as mapping an arbitrarily-sized region of physical memory R with physical base address P and size S bytes, $R = (P, S)$, at an arbitrary virtual base address V. The number of invocations needed to create this simple mapping varies based on V, S, and the desired properties of the mapping (such as page size), as well as the state of the application's virtual address space before the operation. In particular, installing a mapping can potentially entail creating multiple page tables at different levels of the address space in addition to installing a number of page table entries. The library encapsulates the code to do this on demand. In addition, by default, the library uses batch operations within a single page table to amortize system call overhead.

Finally, the library also provides traditional interfaces such as `sbrk()` and `malloc()` for areas of memory where performance is not critical. To simplify start-up, programs running on Barrelfish start up with a limited, conventional virtual address space with key segments (text, data, bss) backed with RAM, though this address space is, itself, constructed by the process' parent using Barrelfish's memory system (rather than the kernel).

It would be straightforward to provide demand paging to disk as in Nemesis [Han99], but Barrelfish does not currently do this. We de-prioritized

demand paging because, firstly, it is largely a matter of engineering rather than a research contribution; secondly, performance-critical applications rely on *not* paging for correctness in time; and thirdly, the growth of non-volatile main memory [PFM15, HP 15] makes it plausible that demand-paging to secondary storage will become irrelevant. Unlike the Linux and Windows memory systems, demand paging on Barrelfish is orthogonal to page size: there is nothing stopping Barrelfish's memory system from demand-paging superpages provided the application has sufficient frames and disk space. Furthermore, the application is aware of the number of backing frames and can add or remove frames explicitly at runtime if required.

The library shows that building a classic VM abstraction over Barrelfish's memory system is straightforward, but the reverse is not the case. Because Barrelfish offers a rich API for constructing page tables from user space, this necessarily means that applications have to keep track of their address spaces to efficiently utilize the API.

We will now discuss the internals of the abstraction implemented in the standard library OS in more detail.

Shadow page tables

The first layer of abstraction provided by the library OS essentially implements shadow page tables which allow the application to keep track of its current address space.

One reason to keep shadow page tables is to alleviate the need to do system calls when doing read-only operations on the address space, such as finding a free virtual region in preparation for creating a new mapping. Further, in order to fully utilize the possibilities we get by using capabilities to allow applications to manage address spaces, we need to have an efficient way to find those page table capabilities, when we want to delete or modify

existing mappings. In addition, we also keep track of more metadata about mappings, such as the mapping capability that gets created during the map system call, the size of each mapping, the capability that is used to back the mapping, etc.

There are two data structures can be considered to implement shadow page tables: linked lists or arrays. To implement shadow page tables with linked lists, we keep a linked list for the entries of each allocated page table, which we keep ordered on the page table entry index. This option, while it is efficient in memory usage, leads to some operations being slower than optimal. For example, looking up a mapping needs up to four linked list traversals instead of four array lookups. Additionally, inserting a new mapping also requires a linked list traversal because we keep our lists sorted by page table entry index.

When implementing shadow page tables using arrays, we consume memory proportional to the number of allocated page tables. Due to the fact that we store quite some metadata for each mapping, this approach can consume more memory than we would like. On the other hand, using arrays has clear performance benefits, as we reduce the complexity of many frequent operations from $O(\#mappings)$ to $O(1)$ by eliminating the linked list traversals. As we can see in figure 3.1, the array-based shadow page table implementation uses about two orders of magnitude more memory. However, to put this in context, the array-based shadow page tables occupy approximately 10 MB for a working set of 4 GB, while the linked-list shadow page tables occupy 100 kB for the same working set size. As even the array-based shadow page tables have less than 1% overhead over of the working set size, we can safely use that approach when we need the advantages such as the $O(1)$ lookups that it brings.

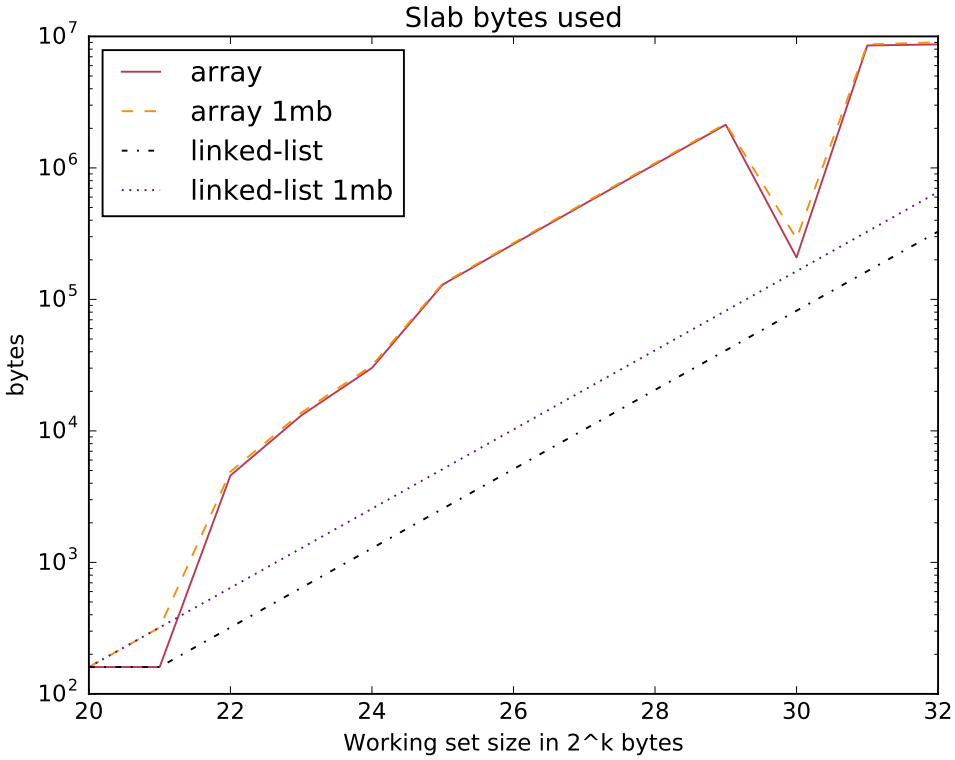


Figure 3.1: *Memory usage for linked list and array-based shadow page table implementations shown in a log-log plot*

Virtual regions and memory objects

On top of the shadow page tables, the library OS presents another layer of abstraction, which is comprised of two distinct types of objects: *virtual regions* and *memory objects*. As mentioned previously, this API is inspired by the Mach memory system.

Virtual regions

Virtual regions give the application a relatively simple API to manage contiguous regions of its virtual address space. While this abstraction does not introduce any new information, it presents information which is available

in the shadow page tables in an easier-to-understand format, if all that the application is worried about is finding a sufficiently large free region in an address space.

The library OS interface for managing virtual address regions consists of two main data structures. First, there is a data structure which represents a contiguous range of virtual addresses in some virtual address space, a *virtual region*. Second, there is a data structure which represents a single virtual address space, and contain references to all the allocated virtual regions in that virtual address space, a *virtual space*. To draw a parallel to the shadow page table abstraction, virtual address spaces and virtual regions can be thought of as a "shadow" virtual address space representation in the respect that modifying the data structures does not actually change the application's address space but just tracks what the application believes to be the layout of its address space(s).

To find free virtual address regions, the virtual region manager uses the shadow page table API. For this to work correctly, each virtual address space is associated with a shadow page table tree which we discussed in section 3.5.2.

Internally the virtual address space data structure keeps a ordered list of virtual regions, however the current implementation does not use this list when allocating fresh regions, opting to directly check the shadow page tables in order to avoid conflicts with potential mappings that the application made directly through the shadow page table API.

Memory objects

While virtual regions are useful to keep track of regions of virtual address space, virtual regions themselves do not know or care about how the region will get backed when a page fault occurs. This is the job of a *memory object*,

which provide a glue layer between `Frame` (and other mappable) capabilities and virtual regions. A virtual region is associated with a memory object, and forwards any page faults to the memory object. The memory object then handles the page fault in a manner which is appropriate to the type of memory object. The default library OS in Barrelfish defines seven different memory objects with different characteristics. The application can also retrieve a semi-opaque handle to a memory object through a virtual region when it desires to modify the mappings backing that virtual region, e.g. to make the region read-only or executable.

Comparison with Mach

As mentioned previously, one of the bigger influences on the user space memory system provided in the default Barrelfish library OS is Mach [RTY+88]. Looking at the layers in Barrelfish's library OS we can see that Barrelfish has picked up the distinction between machine independent and machine dependent parts of the memory system from Mach. Another area where the Mach influence can be seen clearly are the names for some of the different parts of the memory system.

Virtual regions In terms of functionality Barrelfish and Mach virtual regions are mostly identical. A virtual region serves as a handle to an allocated contiguous chunk of virtual address space in both systems.

Memory objects Memory objects are not quite the same. Mach's memory objects are functionally more closely related to Barrelfish Frame capabilities (and other capability types that refer to data), and Barrelfish's memory objects provide an extra layer of abstraction in user space which allows application to create larger memory regions from different Frame

capabilities which can then be treated as a single mappable object in the memory system. It is important to note that – in stark contrast to Mach – memory objects in Barrelfish exist purely at the application level and are not one of the primitives provided by the CPU driver.

Physical map The physical map in Mach is equivalent to the shadow page tables in Barrelfish. The shadow page tables are the part of Barrelfish's user space memory system which is machine dependent as it directly caches the information which exists in the hardware page tables. While Mach treats its pmap layer as a cache that does not always fully track the state of the virtual memory system, Barrelfish's pmaps are generally kept in sync with the virtual memory system, and are exposed to the application which manages the virtual address space which is associated with the pmap.

Pagers Finally, because Barrelfish memory objects are not a kernel primitive, Barrelfish does not have an equivalent mechanism to the pagers which are associated with each Mach memory object. Rather, Barrelfish always reflects a fault to the application which triggered it and leaves handling the fault up to the application itself.

It would be possible to implement something like a pager for an entire Barrelfish application by simply reflecting any faults of an application to a pager instead of the application itself. This would provide a nice mechanism for implementing swapping for potentially non-cooperative applications for which the system would select a trusted swapping service as the application's pager. However, that approach does not fully cover the features available with per-memory object pagers such as elegantly mapping files into memory by just specifying that the pager for the memory object of the memory-mapped file is the file system service.

An approach that would bring Barrelfish closer to Mach in terms of flexibility of handling faults at the expense of increased CPU driver state would be to introduce a new type of capability which more closely models Mach memory objects in their functionality. This capability type could then serve as a replacement for Frame capabilities when creating mappings and faults on such a mapping would be reflected to the new memory object capability's pager. The minimum metadata stored in a memory object capability would be an Endpoint capability to the pager associated with the memory object and the size of the object.

Evaluation

We evaluate Barrelfish by first demonstrating that primitive operations have performance as good as, or better than those of Linux, and then showing that Barrelfish's flexible interface allows application programmers to usefully optimize their systems.

All Linux results, other than those for Dune (Section 3.6.5), are for version 4.2.0, as shipped with Ubuntu 15.10, with three large-page setups: none, `hugetlbfs`, and transparent huge pages. As the Dune patches (git revision 6c12ba0) require a version 3 kernel, these benchmarks use kernel version 3.16 instead. These configurations are summarized in Table 2.3. Thread and memory pinning was done using `numactl` and `taskctl`. Performance numbers for Linux are always the best among all tested configurations.

All experiments, unless specified otherwise, were conducted on an IvyBridge 2x10 core Intel Xeon E5-2670 v2, clocked at 2.5 GHz with 256 GB of RAM. More details of that system are given in table 2.2.

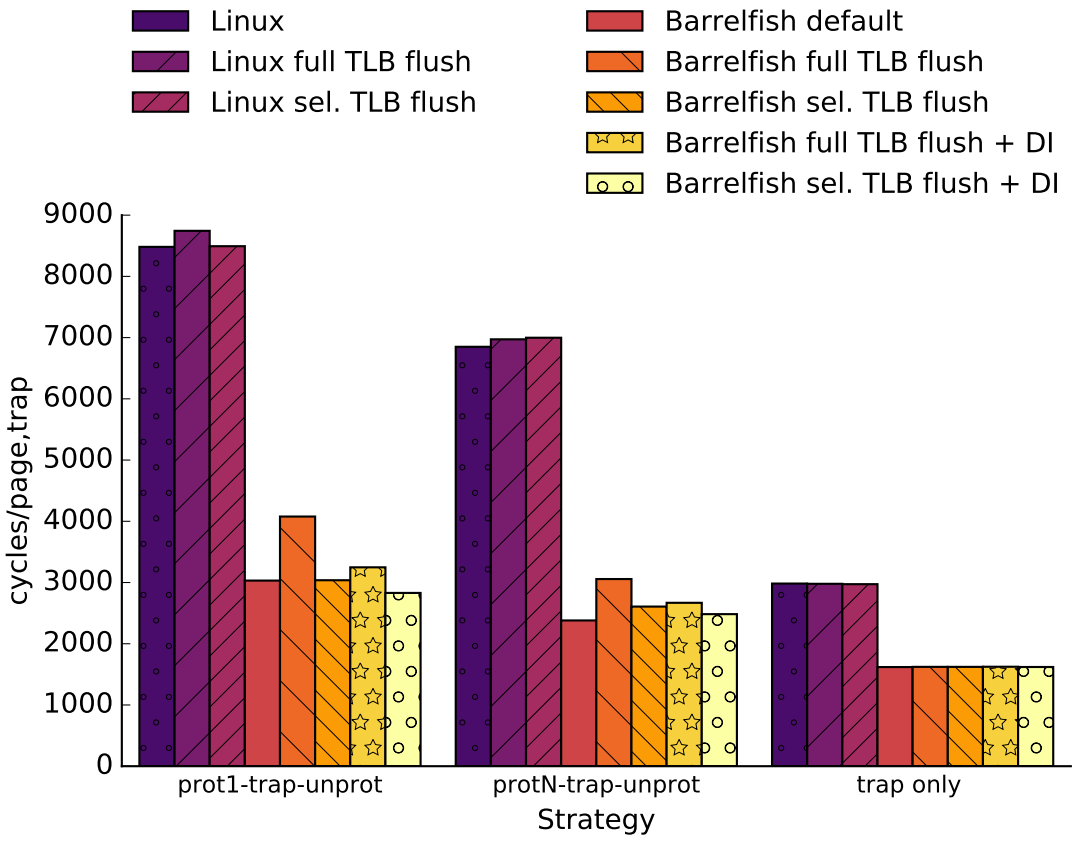


Figure 3.2: *Appel-Li benchmark. (Linux `4.2.0`)*

Appel and Li benchmark

The Appel and Li benchmark [AL91] tests operations relevant to garbage collection and other non-paging tasks. This benchmark is compiled with flags `-O2 -DNDEBUG`, and summarized in Figure 3.2.

We compare Linux and Barrelfish with three different TLB flush modes: 1) Full: Invalidate the whole TLB (writing `cr3` on `x86`) every time, 2) Selective: Only invalidate those entries relevant to the previous operation (using the `invlpg` instruction), and 3) System default: Barrelfish, by default, does a full flush for any operation that involves more than one page. Linux's default behavior depends on kernel version. The version tested (4.2.0) does

a selective flush for up to 33 pages, and full a flush otherwise [Han]. We vary this value to change Linux's flush mode. The working set here is less than 2 MB, and thus large pages have no effect and are disabled.

Barrelfish is consistently faster than Linux here.

For multi-page protect-trap-unprotect (*protN-trap-unprot*), Barrelfish is 64% faster than Linux. For both systems, the default adaptive behavior is as good as, or better than, selective flushing. The Barrelfish *+DI* results use the kernel primitives directly, to isolate the cost of library OS overhead, which is less than 10%.

Memory operation microbenchmarks

We extend the Appel and Li benchmarks, to establish how the primitive operations scale for large address spaces, using buffers up to 64 GB. We *map*, *protect* and *unmap* the entire buffer, and time each operation separately. We compare Barrelfish to the best Linux method for each page size as established in section 2.2.5. On Barrelfish we use the high-level interfaces on a previously allocated frame, for similar semantics to shared memory objects in Linux. Figure 3.3 shows execution time per page.

Map: Barrelfish per-page performance is highly predictable, regardless of page size. Since all information needed is presented to each a system call, the kernel does very little. On Linux we use `shm_open` for 4k pages and `shmat` for others. Linux needs to consult the shared segment descriptor and validate it. This results in a general performance improvement for Barrelfish over Linux up to 15x for 4 kB pages or 93x for large pages, once some upfront overhead is amortized. Barrelfish's upfront overhead, which is quite significant for small buffers, can be attributed to the fact that Barrelfish creates a new kernel object for the mapping, the mapping capability. Creating a capability takes approximately 3000 cycles. This equals 1.2 µs on the machine we use

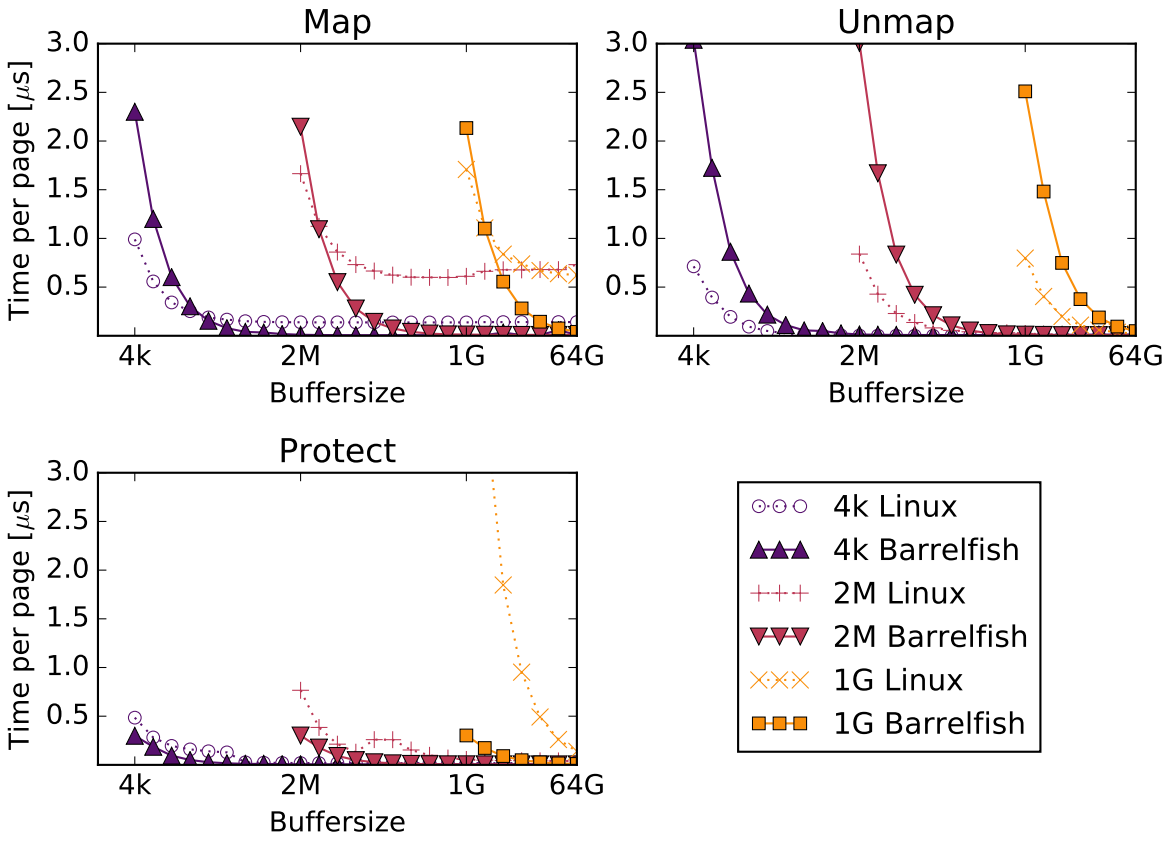


Figure 3.3: *Comparison of memory operations on Barrelfish and Linux using* `shmat`, `mprotect` *and* `shmdt`. *(Linux* `4.2.0-tlbfs`*)*

for the benchmark, and matches the overhead for mapping a 4 kB buffer quite well.

Protect: These results are in line with the Appel and Li benchmarks: Barrelfish outperforms Linux's `mprotect()` on an `mmap`'ed region in all configurations. For large buffers, the differences between Barrelfish and Linux are up to 4x (4 kB pages) or 8x (huge pages).

Unmap: Doing an unmap in Barrelfish is expensive: the high-level interface needs to look up the relevant mapping first, and the actual unmap operation then needs to ensure that all the copies of the mapping capability referring to

the mapping need to be deleted. Linux `shmdt`, however, simply detaches the segment from the process but doesn't destroy it. Barrelfish could be modified to directly invoke the page table, and thereby match the performance of Linux.

Barrelfish memory operations are competitive: capabilities and fast traps allows an efficient virtual memory interface. Even when multiple page table levels are changed, Barrelfish usually outperforms Linux on most cases, despite requiring several system calls.

HPC Challenge RandomAccess benchmark

Many HPC workloads have a random memory access pattern, and spend up to 50% of their time in TLB misses [SGI14]. Using the RandomAccess benchmark [KL] from the HPC Challenge [The] suite, we demonstrate that carefully user-selected page sizes, as enabled by Barrelfish, have a dramatic performance effect.

We measure update rate (Giga updates per second, or GUPS) for read-modify-write on an array of 64-bit integers, using a single thread. We measure working sets up to 32 GB, which exceeds TLB coverage for all page sizes. The Linux configuration is `4.2.0-tlbfs`, with pages allocated from the local NUMA node. If run with transparent huge pages instead, the system always selects 2 MB pages, and achieves lower performance.

Figure 3.4 shows the results on Barrelfish, normalized to 1 GB pages. Performance drops once we exceed TLB coverage: at 2 MB for 4 kB pages, and at 128 MB for 2 MB pages. The apparent improvement at 32 MB is due to exhausting the L3 cache, which slows all three equally, bringing the normalized results closer together. Large pages not only increase TLB coverage, but cause fewer table walk steps to service a TLB miss. Page-structure caches would reduce the number of memory accesses even further but are rather

	Barrelfish		Linux	
Page Size	GUPS	Time	GUPS	Time
4k	0.0122	1397s	0.0121	1414s
2M	0.0408	420s	0.0408	421s
1G	0.0659	260s	0.0658	261s

Table 3.1: *RandomAccess GUPS as a function of page size,* 32 GB *table.*

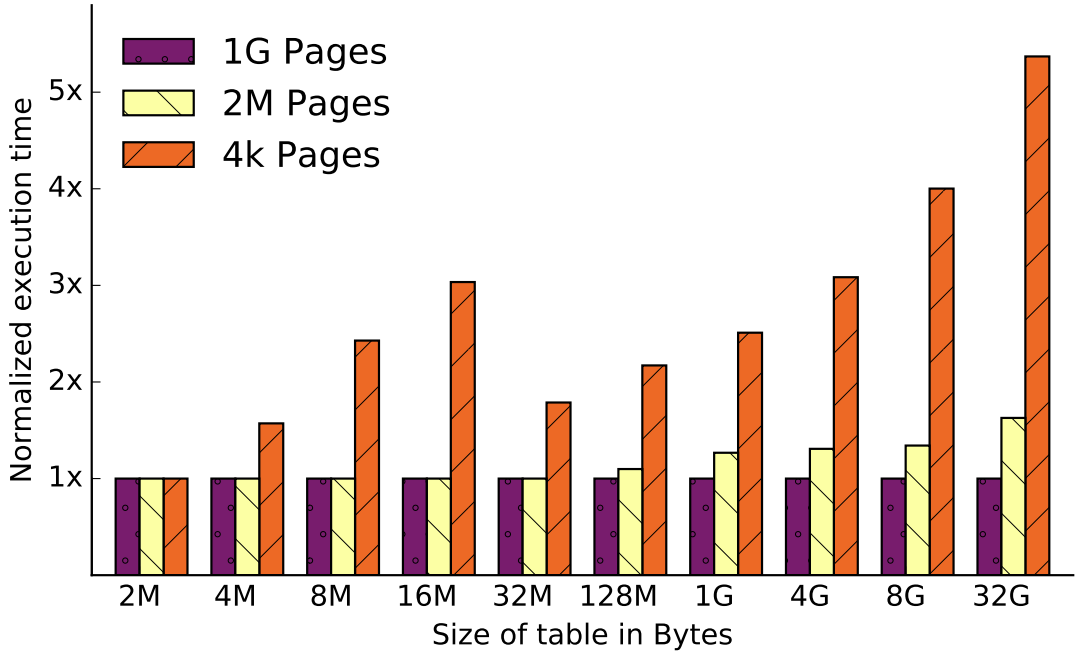


Figure 3.4: *GUPS as a function of table size, normalized, on Barrelfish.*

small [Bha13, BCR10] in size. Barrelfish and Linux perform identically in the test, as Table 3.1 shows. These results support previous findings on TLB overhead [SGI14, BGC+13], and emphasize the importance for applications being able to select the correct page size for their workload.

On Linux, even with NUMA-local memory, high scheduling priority, and no frequency scaling or power management, there is a significant variance between benchmark runs, evidenced by the multimodal distribution in Figure 3.5. This occurs for both `hugetlbfs` and transparent huge pages, and

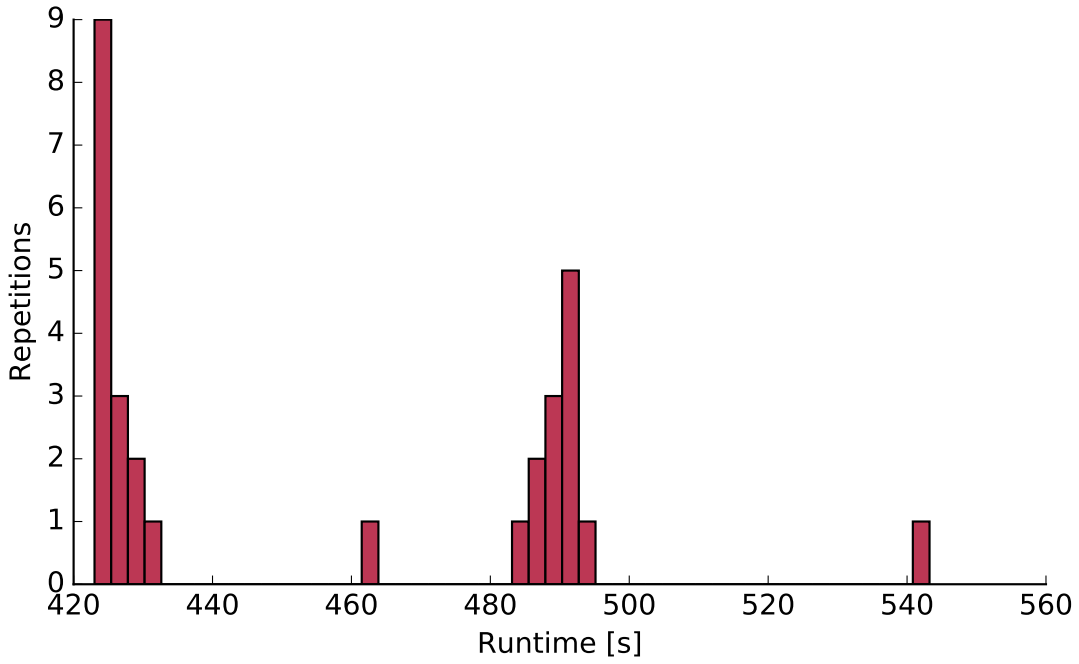


Figure 3.5: *GUPS variance. Linux* `4.2.0-tlbfs`*,* 2 MB *pages.*

is probably due to variations in memory allocation, although we have been unable to isolate the precise cause. This variance is completely absent under Barrelfish even when truly randomizing paging layout and access patterns, demonstrating again the benefit of predictable application-driven allocation.

Mixed page sizes

Previous work [GLD+14] has shown that while large pages can be beneficial on NUMA systems, they can also hurt performance. Things are even more complicated when there are more page sizes (e.g., 4 kB, 2 MB, 1 GB for `x86_64`). Furthermore, modern machines often have a distinct TLB for each page size, suggesting that using a mix of page sizes increases TLB coverage.

Kaestle et al. [KARH15] showed that distribution and replication of data mitigates congestion on interconnects and balances memory controller load,

by extending Green-Marl [HCSO12], a high-level domain-specific language for graph analytics, to automatically apply these techniques per region, using patterns extracted by the compiler. This gave a two-fold speedup of already tuned parallel programs.

Large pages interact with the NUMA techniques described above, by changing the granularity at which they can be applied to data structures that are contiguous in virtual memory. The granularity of NUMA distribution, for example, is the page size. Hence, the smaller the page size the more slack the run-time has to distribute data across NUMA nodes. Bigger page sizes also make memory allocation more restrictive: The starting address when allocating memory must be a multiple of the page size. Bigger page sizes can increase fragmentation and increases the chance of conflicts in caches and TLB.

In Barrelfish, programs map their own memory, and all combinations of page sizes are supported. Furthermore, no complex setup of page allocations and kernel configurations are required.

Table 3.3 shows the effect of the page size on application performance using Shoal's Green-Marl PageRank [KARH15]. NUMA effects are minimal on the 2-socket machine we are using in other experiments, so for this experiment we use the machine in Table 3.2 and note that AMD's SMT threads (CMT) are disabled in our experiments.

We evaluate two configurations: First, single-threaded (T=1). In this case replication does not make sense as all accesses are local, and distribution is unnecessary as a single thread cannot saturate the memory controller — indeed, an increase in remote memory access would likely reduce performance. In this case, an isolated application, bigger pages are always better.

Next, we run on all cores and explore the impact of replication and distribution on the choice of page sizes. 1 GB pages clearly harm performance

CPU	AMD Opteron 6378
micro architecture	Piledriver
#nodes / #sockets / #cores	8 / 4 / 32 @ 2.4 GHz
L1 / L2 cache size	16 kB / 2 MB per core
L3 cache size	12 MB per socket
dTLB (4 kB pages)	64 entries, fully
dTLB (2/4 MB pages)	64 entries, fully
dTLB (1 GB pages)	64 entries, fully
L2 TLB (4 kB pages)	1024 entries, 8 way
L2 TLB (2/4 MB pages)	1024 entries, 8 way
L2 TLB (1 GB pages)	1024 entries, 8 way
RAM	512 GB (64 GB per node)

Table 3.2: *Specification of machine used in §3.6.4*

page size	array configuration		
	T=1	T=32 (dist)	T=32 (repl + dist)
4 kB	597.91	**51.32**	34.43
2 MB	414.80	58.09	**28.87**
1 GB	**395.64**	265.94	128.77

Table 3.3: *PageRank runtime (seconds) depending on page size and PageRank configuration (repl = replication, dist = distribution, T is the number of threads). Highlighted are best numbers for each configuration. Standard error is very small.*

as distribution is impossible or too coarse-grained. We only break even if 90% of the working set is replicated. However, the last 10% still cannot be distributed efficiently, which leads to worse performance.

It is clear that the right page size is highly dynamic and depends on workload and application characteristics. It is impractical to statically

configure a system with pools (as in Linux) optimally for all programs, as the requirements are not known beforehand. Also, memory allocated to pools is not available for allocations with different page sizes. In contrast, Barrelfish's simpler interface allows arbitrary use of page sizes and replication by the application without requiring *a priori* configuration of the OS.

Page status bits

The potential of using the MMU to improve garbage collection is known [AL91]. Out of many possible applications, we consider detecting page modifications; A feature used, for example, in the Boehm garbage collector [BDS91] to avoid stopping the world. Only after tracing does the collector stop the world and perform a final trace that need only consider marked objects in dirty pages. This way, newly reachable objects are accounted for and not collected.

There are two ways to detect modified pages: The first is to make the pages read-only (e.g., via `mprotect()` or transparently by the kernel using soft-dirty PTEs [sof]), and handle page faults in user-space or kernel-space. The handler sets a virtual dirty bit, and unprotects the page to allow the program to continue. The second approach uses hardware dirty bits, set when a page is updated. Some OSes (e.g., Linux) do not provide access to these bits. This is not just an interface issue. The bits are actively used by Linux to detect pages that need to be flushed to disk during page reclamation. Other OSes such as Solaris expose these dirty bits in a read-only manner via the `/proc` file-system. In this case, applications are required to perform a system call to read the bits, which, can lead to worse performance than using `mprotect()` [Boea].

In Barrelfish, physical memory and page tables are directly visible to applications. Applications can map page tables read-only in their virtual address

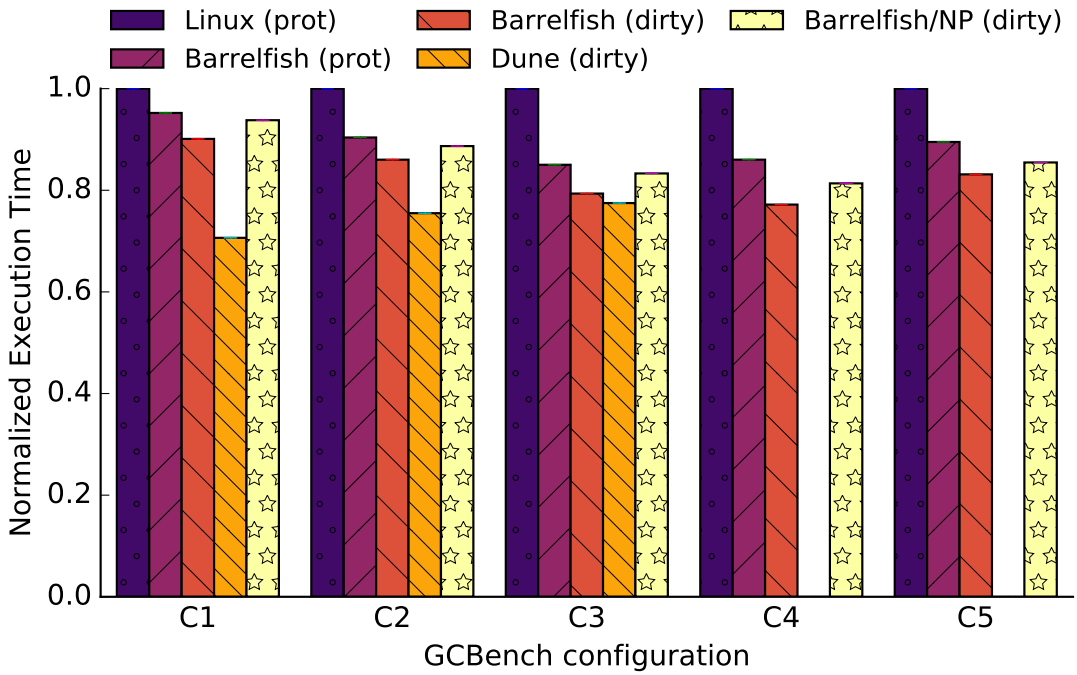


Figure 3.6: *GCBench on Linux, Barrelfish and Dune, normalized runtime to Linux. (Linux* `3.16`*,* `3.16-dune`*)*

space. Only clearing the dirty bits requires a system call.

Dune [BBM+12] provides this functionality through nested paging hardware, intended for virtualization, by running applications as a guest OS. Dune applications have direct access to the virtualized (nested) page tables. This approach avoids any system call overhead to reset the dirty bits, but depends on virtualization hardware and can lead to a performance penalty due to greater TLB usage [BGC+13, BSSM08].

We use the Boehm garbage collector [BDS91] and the GCBench microbenchmark [Boeb]. GCBench tests the garbage collector by allocating and collecting binary trees of various sizes. We run this benchmark with the three described memory systems, Linux, Dune and Barrelfish with five different configurations C1 to C5, which progressively increase the size of the allocated trees.

Config	C1	C2	C3	C4	C5
Runtime (s)					
Linux (prot)	2.1	9.6	42	191	848
Barrelfish (prot)	2.0	8. 7	37	166	760
Barrelfish (dirty)	1.9	8.3	34	149	705
Dune (dirty)	1.5	7.3	33	–	–
Barrelfish/NP (dirty)	2.0	8.6	36	157	720
Collections					
Linux (prot)	251	336	381	428	448
Barrelfish (prot)	247	330	385	430	444
Barrelfish (dirty)	231	325	383	436	443
Dune (dirty)	318	367	403	–	–
Barrelfish/NP (dirty)	233	325	381	434	443
Heap size (MB)					
Linux (prot)	139	411	1924	7972	24932
Barrelfish (prot)	139	475	1515	6951	27486
Barrelfish (dirty)	105	475	1481	5911	28995
Dune (dirty)	106	386	1579	–	–
Barrelfish/NP (dirty)	100	453	1573	5541	28132

Table 3.4: *GCBench reported total runtime, heap size and amount of collections.*

In Figure 3.6 we compare the runtime of each system. Barrelfish implements all three mechanisms: protecting pages (Barrelfish (prot)), hardware dirty bits (Barrelfish (dirty)) in user-space and hardware dirty bits in guest ring 0 (Barrelfish/NP (dirty)) (as does Dune). Our virtualization code is based on Arrakis [PLZ+14].

Barrelfish (prot) performs between 4% (C1) and 13% (C4) better than Linux (prot). This is consistent with Figure 3.3 where Barrelfish performs

better than Linux for protecting a single 4 kB page. We further improve Barrelfish's performance (between 4% (C2) and 10% (C4)) when we use hardware dirty bits, by avoiding traps when writing to pages. We still incur some overhead as we have to make a system call to reset the dirty bits on pages. Dune outperforms Barrelfish (dirty) by up to 21% (C1), as direct access to the guest page tables enables resetting the dirty bits without having to make a system call. However, Barrelfish manages to close the gap as the working set becomes larger, in which case Dune performance noticeably shows the overhead of nested paging. Unfortunately, we were unable to get Dune working with larger heap sizes on our hardware and thus have no numbers for Dune for configurations C4 and C5.

On Linux, using transparent huge pages did not have a significant impact on performance and we report the Linux numbers with THP disabled. In a similar vein, we were unable to get Dune working with superpages, but we believe that having superpages might improve Dune performance for larger heap sizes (c.f. 3.6.3).

Barrelfish/NP (dirty) runs GCBench in guest ring 0 and reads and clears dirty bits directly on the guest hardware page tables. The performance for Barrelfish/NP is similar to Barrelfish (dirty) and slower than Dune. However, this can be attributed to the fact that Barrelfish/NP does not fully leverage the advantage of having direct access to the guest hardware page tables and still uses system calls to construct the address space.

Table 3.4 shows the total runtime, number of collections the GC did and the heap size used by the application. Ideally, the heap size should be identical for all systems since it is always possible to trade memory for better run time in a garbage collector. In practice this is very difficult to enforce especially across entirely different operating systems. For example Barrelfish uses less memory (25%) for C4 compared to Linux (prot) but more memory (14%) for C5.

We conclude that with Barrelfish we can safely expose MMU information to applications which in turn can benefit from it without relying on virtualization hardware features.

Nested paging overhead

To illustrate the potential downside of nested paging, we revisit the HPC Challenge RandomAccess benchmark. Resolving a TLB miss with nested paging requires a 2D page table walk and up to 24 memory accesses [AJH12] resulting in a much higher miss penalty, and the overhead of nested paging may end up outweighing the benefits of direct access to privileged hardware in guest ring zero. The RandomAccess benchmark represents a worst-case scenario due to its lack of locality.

We conduct the same experiment as in section 3.6.3 on Dune [BBM+12] with a working set size ranging from 1 MB to 128 MB. Figure 3.7 and Table 3.5 show that for the smallest table sizes (1 MB and 2 MB) the performance of RandomAccess under Dune and Linux is comparable. Larger working set sizes exceed the TLB coverage and hence more TLB misses occur. This results in almost 2x higher runtime for RandomAccess in Dune than Linux. As for all comparisons with Dune, we disable transparent huge pages on Linux.

Running applications in guest ring zero as in Dune has pros and cons: on one hand, the application gets access to privileged hardware features, on the other hand, the performance may be degraded due to larger TLB miss costs for working sets which cannot be covered by the TLB.

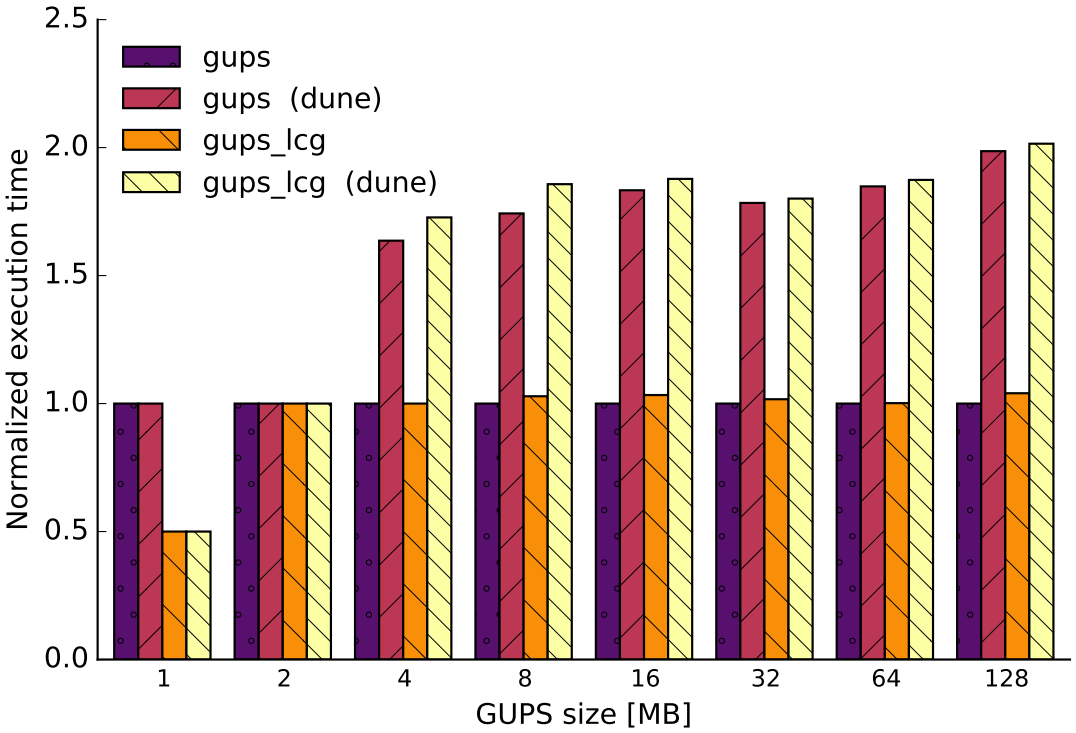


Figure 3.7: *Comparison of the execution time of RandomAccess with and without nested paging for varying working set sizes, normalized to GUPS on native Linux. (Linux `3.16`, `3.16-dune`)*

Page coloring

The core principle of paged virtual memory is that virtual pages are backed by arbitrary physical pages. This can adversely affect application performance due to unnecessary conflict misses in the CPU caches and an increase in non-determinism [KKAE11]. In addition, system wide page coloring introduces constraints on memory management which may interfere with the application's memory requirements [ZDS09].

Implementing page placement policies is non-trivial: The complexity of the FreeBSD kernel is increased significantly [Dil13], Solaris allows applications to chose from multiple algorithms [Ora10], and there have been several

Size (MB)	Linux GUPS	Linux GUPS LCG	Dune GUPS	Dune GUPS LCG
1	2	1	2	1
2	3	3	3	3
4	11	11	18	19
8	35	36	61	65
16	90	93	165	169
32	236	240	421	425
64	594	595	1098	1113
128	1510	1571	2999	3043

Table 3.5: *RandomAccess absolute execution times in milliseconds. (Linux `3.16`, `3.16-dune`)*

failed attempts to implement page placement algorithms in Linux. Other systems like COLORIS [YWCL14] replace Linux' page allocator entirely in order to support page coloring.

In contrast, Barrelfish allows an application to explicitly request physical memory of a certain color and map it according to its needs. For instance, a streaming database join operator can restrict the large relation (which is streamed from disk) to a small portion of the cache as most accesses would result in a cache miss anyway and keep the smaller relation completely in cache.

Table 3.6 shows the results of parallel execution of two instances of the HPC Challenge suite RandomAccess benchmark on cores that share the same last-level cache. In the first column we show the performance of each instance running in isolation. We see a significant drop in GUP/s for the instance with the smaller working set when both instances run in parallel. By applying cache partitioning we can keep the performance impact on the smaller instance to a minimum while improving the performance of the

Process	Isolation	Parallel		Parallel Colors	
16M Table	0.0926	0.0834	90.0%	0.0921	99.5%
64M Table	0.0570	0.0561	98.4%	0.0631	110.7%

Table 3.6: *Parallel execution of the RandomAccess benchmark on Barrelfish with and without cache coloring. Values in GUPS.*

larger instance even compared to the case where the larger instance runs in isolation.

The reason behind this unexpected performance improvement is that the working set (the table) of the larger instance is restricted to a small fraction of the cache which reduces conflict misses between the working set and other data structures such as process state etc.

Discussion

With this evaluation, we have shown that the flexibility of Barrelfish's memory system allows applications to optimize their physical resources for a particular workload independent of a system-wide policy without sacrificing performance.

Barrelfish's strength lies in its flexibility. By stripping back the policies baked into traditional VM systems over the years (many motivated by RAM as a scarce resource) and exposing hardware resources securely to programs, it performs as well as or better than Linux for most benchmarks, while enabling performance optimizations not previously possible in a clean manner.

4

A protocol for decentralized capabilities

The memory system design presented in the previous chapter has not discussed scalability at all. In this chapter we describe a protocol for a distributed capability system and its implementation in Barrelfish. This protocol is designed in such a fashion to allow capability operations to scale to a large multicore machine without introducing large operation latencies in most cases. To reiterate, as Barrelfish's capabilities support five operations: copy, retype, delete, revoke, and invoke, our protocol needs to support each of those operations on objects for which capabilities exist on multiple cores in the system. This protocol has previously been discussed by Mark Nevill in his master's thesis [Nev12]. We claim that a scalable capability system is sufficient to allow Barrelfish's memory system to scale. We believe this claim to be true because the memory system is implemented entirely on top of the capability system.

Overall design

Our distributed capability system is designed around nodes (e.g. cores in Barrelfish) which each hold a partial replica of the capability database. We specify how each capability operation interacts with these replicas in section 4.2. Our protocol eliminates the need for elaborate agreement protocols by relying on two assumptions: 1) we assume that message channels between nodes are strictly FIFO and no messages get lost, and 2) nodes are not adversarial. Barrelfish fulfils both assumptions. Barrelfish's IPC channels have preserve message order and guarantee exactly-once delivery if the send operation succeeds. Send can fail explicitly and Barrelfish puts the burden of retrying to send on the application. As we implement the protocol in the *monitors* which are part of Barrelfish's trusted computing base, we can assume that nodes are not adversarial and can be trusted to carefully ensure that messages are sent.

We design a distributed protocol which picks an arbitrary node for each capability which serves as the serialization point for operations on that capability and uses the assumptions we make to simplify synchronization. We call that node the *owner* of the capability. Further we call a capability copy residing on it's owner node *local*, and all copies on nodes other than the capability's owner *foreign*.

Our protocol has three invariants. The first invariant is that any capability that exists is required to have an owner.

Invariant 4.1. *Each non-null capability has an owner node.*

Invariant 4.1 by itself does not prohibit a capability from having multiple owners. Thus, we need another invariant to ensure that for any capability in the system there is a single point of serialization.

Invariant 4.2. *Any two capabilities that are copies must have the same owner node.*

Additionally, to simplify all protocol operations we require the owning node to hold at least one copy of each capability for which it is the owner. Or, in other words, for each capability there is at least one local copy.

Invariant 4.3. *For each capability, there is at least one local copy.*

Capability operations

We will first give a definitions of each capability operations for a distributed system of nodes which each keep an index into their partial local replica of the capability database.

Before going into the specifics for each operation, 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 "$\leftarrow$", we copy the capability metadata into the destination slot and update the capability database replica 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. To ensure that the ownership invariant (invariant 4.2) is not violated, it is important that the copy operation creates the new copy and defines the new copy's owner in a *single and atomic*[1] step. The implementation in Barrelfish

[1] with respect to the distributed capability protocol

uses the fact that the CPU driver is non-preemptable to construct atomic operations for the capability protocol.

Algorithm 1 copy

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

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 which were not explicitly requested, the target core must also become the owner of the new capabilities. The implementation of this protocol in Barrelfish allows the creation of multiple non-overlapping sub-ranges in a single retype operation akin to batching page table manipulations as discussed in the previous chapter. For the sake of simplicity, we only create one output capability per retype in the protocol specification, allowing retype operations to specify a single sub-region of the source capability that is used for the output capability. The implementation in Barrelfish is equivalent to performing multiple single-region retypes in the same transaction.

Algorithm 2 retype

function RETYPE(*cap*: slot, *region*: range, *type*: captype, *dest*: slot)
  if *dest* is not `Null` $\vee$ *retype*(*cap*, *region*, *type*) is not valid **then**
    fail the RETYPE operation.
  if any conflicting descendants exist locally or remotely **then**
    fail the RETYPE operation.
  begin atomic
    dest $\leftarrow$ local *retype*(*cap*, *region*, *type*) on *location*(*dest*).
    set *dest* to "local".
  end atomic

Delete We distinguish between deleting local and foreign capabilities, as the amount and type of work we need to do is very different for these two cases.

Deleting a local copy of a capability can get complicated because the owning core must always have a copy of the capability, cf. protocol invariant 4.3. Therefore when we delete the last copy of a capability on the owning core, and other copies of the capability still exist in the system, we must transfer ownership to a core that still holds at least one copy.

This is further complicated because not all capability types support changing ownership: capabilities of some types, e.g. CNode and Dispatcher capabilities, represent kernel state, and would require synchronization outside of the capability system, if we want to migrate them from one CPU driver to another.

As shown below, in algorithm 3, deleting a foreign copy is trivial, we just set the capability slot which holds the capability we wish to delete to `Null`.

Algorithm 3 delete

```
function DELETE(cap: local slot)
    begin atomic
        if last copy on owner(cap) then
            if cap is not moveable then
                for all foreign copies on all cores do
                    DELETE (copy).
                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).
        cap ← NULL
    end atomic

function DELETE(cap: foreign slot)
    cap ← NULL
```

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

```
function CHOWN(cap: slot)
    begin atomic
        for all copies of cap on all cores do
            set owner(copy) to core(cap).
    end atomic
```

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. We discuss the finer details of distributed deletes in section 4.3.

Algorithm 5 revoke

```
function REVOKE(cap: local slot)
    begin atomic
        for all immediate descendants on all cores do
            REVOKE descendant.
            DELETE descendant.
        for all copies on all cores do
            DELETE copy
    end atomic

function REVOKE(cap: foreign slot)
    CHOWN (cap).
    REVOKE (cap).
```

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 impacted by the delete, and so the

capability slot can be cleared by the kernel directly without need for any cross-core negotiation.

In the second 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 initially requested delete is transformed to 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 kernel 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 any 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 back of a singly-linked "delete" list stored within the extended region of the capability slot.

3. Work through the "delete" list, performing a "delete" as described above on every slot contained in the objects referenced by the list entry. Then place the entry at the front of a "clear" list.

4. Walk through the clear list, performing the final clean-up of every entry in the list.

It is vital that the clear list is treated as a stack, as otherwise there is a possibility that we would erase part of the clear list itself by doing a clear step. Consider the following example: the first element of the clear list is a CNode, *cn1*, which itself contains a further CNode capability *cn2*. In this case the pointer from the first clear list element, *cn1*, would point into the memory region referred to by *cn1* which would lead to a dangling pointer after cleaning up *cn1* and removing *cn1* from the clear list.

The corresponding capability state machine is shown in Figure 4.1. Copy and retype are not included; both operations simply put the capability in the locked state until the operation completes or fails. This is required to ensure operation atomicity from the perspective of the rest of the system.

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.

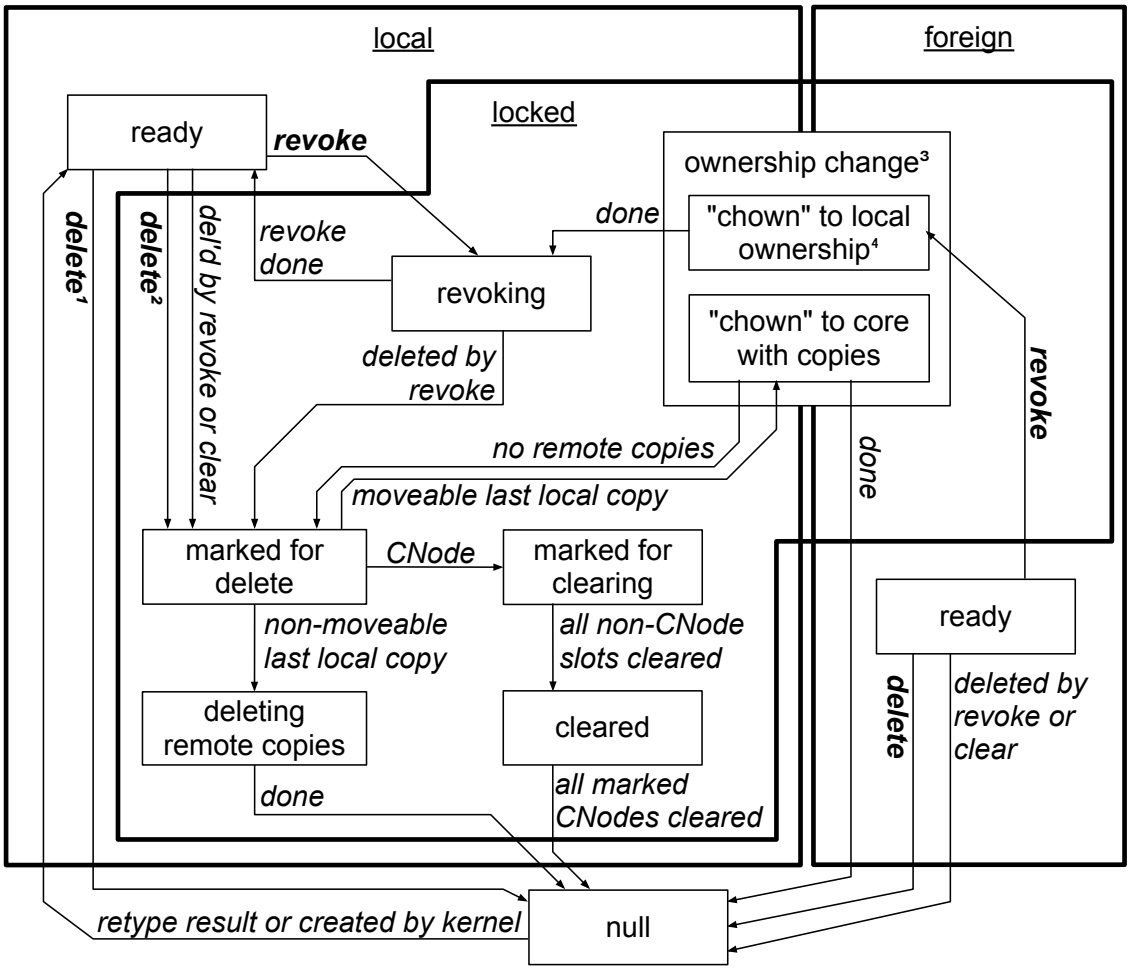


Figure 4.1: *Per-capability slot state machine for deletes and revokes. Actions in bold are user-initiated.* [1,2] *When local copies of a capability exist, delete can directly null the capability slot.* [3] *When changing ownership, the invariant of having exactly one owning core that is equal for all copies in the system may be temporarily violated.* [4] *This may fail, returning the capability slot to the ready state.*

Capability transfer

To perform the operations described in section 4.2 we need to make sure that nodes in the system can exchange information about capabilities that they hold. In particular, the nodes need to be able to share and exchange the metadata stored in each capability. (NB: This, of course, necessitates that the software component that deals with distributed capability operations on each node has to be trusted.) To enable nodes to exchange capability data, we must define a serialization protocol which the sender can use to produce a byte array given a capability, and the receiver can use to recover a capability from the received byte array. The receiver then needs to ensure that the new capability can be used on the node. To do this, the receiver needs to register the new capability in the node's mapping database. The mapping database needs to support some new query types to enable the receiver to efficiently insert a new capability into the local mapping database.

Receiving a copy First, upon receiving a serialized capability during a copy operation, the receiver needs to insert the new capability into the mapping database. Because the mapping database is used to look up capability relations, inserting a received copy needs to work without prior knowledge of copies, ancestors, or descendants on the receiving node.

Retype checks Second, checking if a capability retype operation is legal requires checking if the requested region already has conflicting descendants on any node in the system. Again, the receiving node has to perform the retype checks with only a serialized copy of the source capability without knowledge of any pre-existing local relations. One way to deal with this would be to temporarily insert the capability into the receiving node's mapping database, check for descendants in the requested region, and then

delete the temporary capability again. However, this approach creates – for a brief time – a capability that does not appear in the high-level description of the retype operation. While this might not actually impact the system, we try to avoid creating such temporary copies whenever possible, and therefore provide support for querying whether a region of a capability has descendants without having to insert the capability into the mapping database.

Deleting last owned copy Third, deleting the last copy of a capability on the owning node requires that the system either finds another copy of the capability or recognizes that the last copy in the system is being deleted. Again, creating a temporary copy to perform this check is undesirable, especially because in this case we are trying to ascertain the existence of copies. Therefore we need a way to query a mapping database for copies without inserting the capability we are searching for.

Reclaiming memory regions Fourth, to not leak physical memory(!), we need the ability to find regions of physical memory for which no capability exists in the system. If we implement eager memory reclamation when deleting a capability, determining whether we are actually deleting the last capability referring to a region of memory requires a system-wide search for ancestors, copies and descendants. Copies, in this case, are tracked by the algorithm described in 4.2, and we already covered the check for descendants previously when discussing the query requirements for retype. However, reclaiming memory eagerly when deleting a capability also requires checking for ancestors. In particular, this check for ancestors must be performed without creating visible copies of the capability which is being deleted.

Alternatively, we can defer reclaiming memory, and periodically scan the entirety of the system's memory for regions which are not covered by any

capability. The minimum requirements for this operation are that we know about all the physical memory in the system and the ability to search the system for the "first" capability, and forward or backward siblings of a given capability.

Revoke Revoke deletes all copies and descendants of a capability. We can reliably find copies and descendants on the node where the revoke is executed, as revoke can only be executed on the capability's owning node. However, we need to find descendants of the capability even on nodes where no copies exist. To do this we need a way to search for capabilities covering a given region, as already discussed for the retype case.

Implementing a mapping database

As mentioned earlier in this dissertation, we have a database of all capabilities on a core which allows the CPU driver to quickly find capabilities by relation, such as finding a copy or descendant of a capability. We call this database mapping database.

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

However, a preorder DFS linked-list mapping database containing n capabilities requires a $O(n)$ linear scan when inserting a capability for which we do not know any relations beforehand, as does finding ancestors and descendants given only a capability's value. As these operations are performed in the non-preemptable privileged kernel code, any operation that requires

$O(n)$ time creates a scheduling hole of problematic size, especially for latency sensitive applications, or applications with real-time requirements.

We show the median operation latencies for the doubly-linked list mapping database in figure 4.2. As discussed we see that inserting an element into a doubly-linked list without a pointer to a close neighbor is fairly expensive and has highly unpredictable latency, as evidenced by the standard deviation of more than thirty thousand cycles. Additionally, checking whether any ancestors exist for a capability takes more than twenty thousand cycles.

All the microbenchmarks shown in this section were performed on a 2x10 Intel Xeon E5-2670 v2 clocked at 2.5 GHz.

To avoid operations with linear complexity in the number of capabilities in the mapping database, we will replace the linked list with a more suitable search data structure.

Review of search data structures

Another popular data structure for lookup-heavy workloads is the hash table. Hash tables provide $O(1)$ lookup with high probability and $O(1)$ insertion, either amortized or with high probability. However for our use case we require several properties that a hash table cannot provide in an efficient manner. Firstly, hash tables cannot directly represent the hierarchical relationships between capabilities. Therefore we would need to maintain additional metadata to efficiently find immediate ancestors and descendants. Because copies of capabilities can disappear, it is not enough to simply keep track of a capability's ancestor and descendants by keeping a pointer to each. Rather, those pointers would need to be checked and updated on every delete. Secondly, while a hash table's space complexity is $O(n)$ there is no direct relationship between the memory used by the table and the elements stored in the table. Because of this, one would need to dynamically allocate

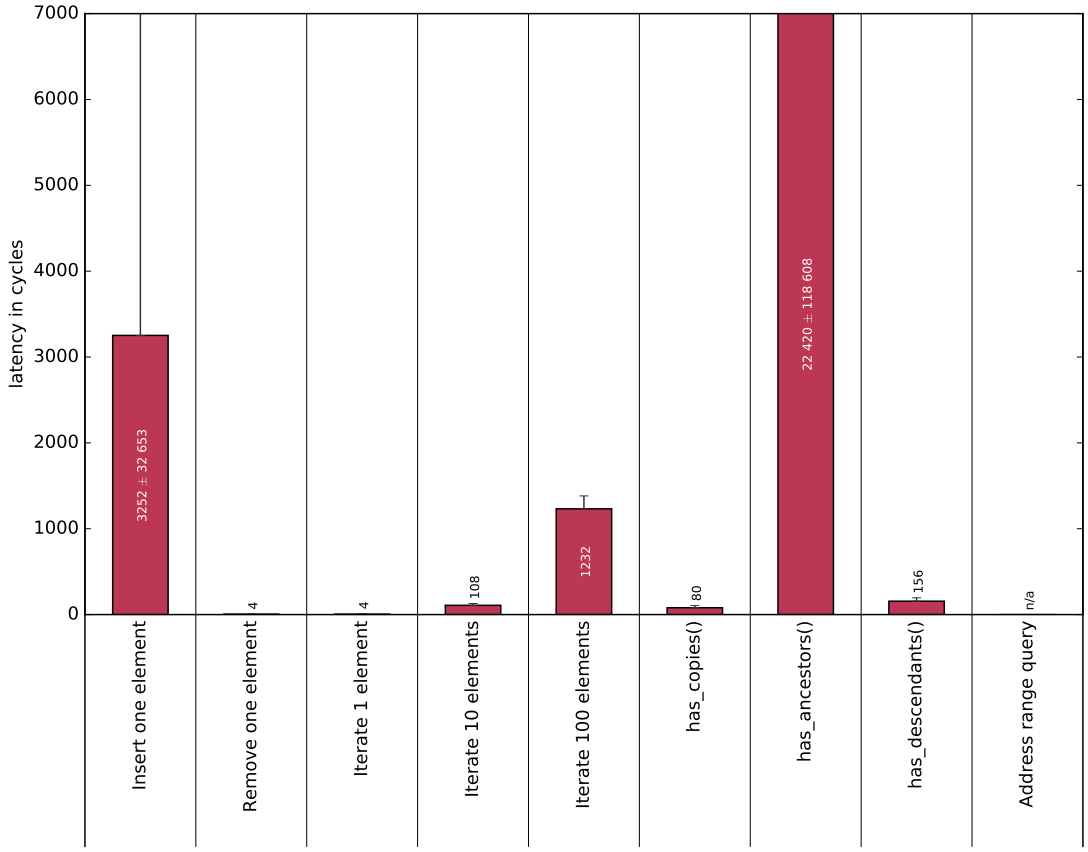


Figure 4.2: *50 percentile latency for a number of operations on a doubly-linked list implementation of a mapping database. The database contains 4096 capabilities.*

memory outside of the capabilities, that is, outside the CNodes. However, the mapping database is stored and maintained by the CPU driver, and, as mentioned previously, one reason for using the capability model is precisely to avoid such allocations in the CPU driver.

iThe next approach we consider is to design 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 to be merged into the deleted node's 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 its 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 dynamically allocate space external to capabilities.

To circumvent this problem, we simply avoid directly storing the relationships. 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, i.e. the CNodes, 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 binary search tree to an interval tree using the augmentation technique described in Cormen et al.[CLRS01, p. 311 – 317], which allows us to search for capabilities covering a address range, which we employ for the ancestor query and the region queries necessary for retype and memory reclamation.

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[Knu73]. 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 on a 64-bit architecture, and 24 bytes on a 32-bit architecture. Every capability must be able to store a tree node, so we have to reserve this space in every capability slot, regardless of whether the tree node in a given slot is used or not at any point in time.

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. This eliminates the need for node

migration entirely. 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 [And93]. 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.

Ordering

To implement a tree-based index, we need to define an ordering on the items we want to index. 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 sub-region 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: all copies 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 traversed 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.

Range Queries

By augmenting the tree with an end interval as described in Cormen at al.[CLRS01, p. 311 – 317] we gain the ability to perform searches for ranges. The storage cost for each tree node increases by 9 bytes, as we need to store a 64-bit address and a 8-bit type root indicating the largest address covered by the subtree rooted at the tree node. Note that looking up capabilities for a single address is also a range search, as an address may be "covered" by 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 as discussed in chapter 3.

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.

Augmented AA tree implementation trade-offs

Because we have a hard limit of 64 bytes[2] per capability slot, and the capability type specific part of each slot consumes 24 bytes, we are left with a maximum of 40 bytes to store a capability's metadata.

Assuming that we will always have 64 bytes available per capability, we quickly run into space problems on machines with 64-bit architectures. For the rest of this section, we skip the calculations with 32-bit pointers, as we assume that all the required metadata would fit comfortably in 40 bytes, i.e. 10 pointers.

Of the 40 bytes that are left after storing the actual capability data, the augmented AA tree node requires three pointers, a capability type, and the address and type of the largest address covered by the subtree. This is $3 \cdot 8 + 1 + 8 + 1 = 34$ bytes. This leaves us with just 6 bytes to (1) keep track of the capability's owning core, (2) cache information about potential remote ancestors, copies, and descendants, (3) the capability's lock, (4) a flag indicating whether a capability is currently being deleted, and (5) a pointer for the queue which is required to keep track of the capability if it is part of a delete cascade as discussed in section 4.3.

Currently, we need 8 bits to store a core identifier, 1 bit each to cache whether a capability has remote ancestors, copies, or descendants, 1 bit each

[2]We can – and did for a while – make capability slots bigger, but prefer to keep them cache-line sized

for the capability's lock and delete state, and 64 bits for the delete queue pointer. Summing up, this gives us a requirement of $8+3\cdot 1+1+1+64 = 77$ bits, or, rounded up, 10 bytes.

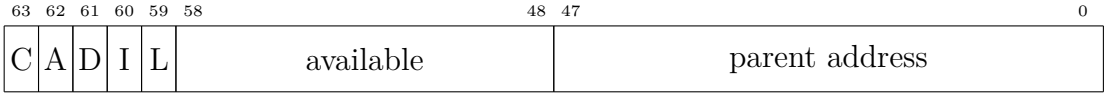
Summing up the space requirement for all the metadata, we see that we exceed the 40 available bytes by 4 bytes. We will now discuss a two strategies to optimize the size of a capability's metadata such that all the information that needs to be stored fits into 64 bytes. For this discussion, we assume that the actual capability part cannot be shrunk further, and would prefer to keep some spare bytes for the eventuality of the actual capability part requiring more space in the future.

The first option we consider is to drop the parent pointer from tree nodes so we can use the space for other purposes, reducing the tree node size to $2 \cdot 8 + 1 + 8 + 1 = 26$ bytes for the augmented version of the tree. Dropping the parent pointer 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 second strategy is to use the unused high bits of a virtual address – current 64-bit architectures have a maximum usable virtual address size of 48 bits – to store the small metadata items in order to keep the total size of a capability including metadata below, or at, 64 bytes.

We employ the following layouts for storing the small, that is, 8 bits and smaller, metadata items in the high bits of the three tree pointers.

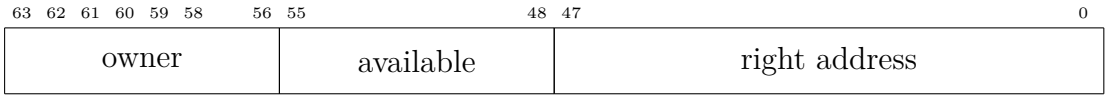
We store all the one bit metadata in the highest 5 bits of the parent pointer.



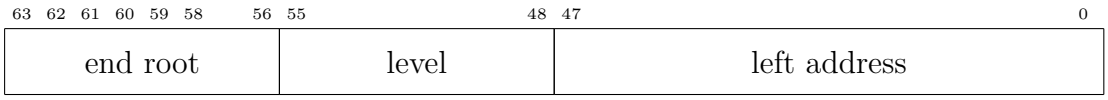
The characters C,A and D identify the bits occupied by the flags indicating the presence of remote copies, remote ancestors and remote descendants

respectively. The character I identifies the bit storing the flag indicating that the capability is currently being deleted. Finally, the character L identifies the bit representing the capability's lock state.

We store the owning core's identifier in the high 8 bits of the right pointer. We intentionally leave the next 8 bits unused, to account for future systems where we may want to identify more than 256 distinct owners.



Finally we store the 1 byte values giving the tree node's *level* and *end root* in the high bytes of the left pointer. Note that those two values are only required by the tree implementation, which is one reason why we co-locate them in the same pointer.



With this layout, we are able to reduce the size for the augmented AA tree node by 4 bytes, and therefore manage to just squeeze a full capability into $24 + 4 \cdot 8 + 8 = 64$ bytes.

We evaluate two variants of implementing this layout, one implementation uses masks and bit shifts to read, write and mask the high bits of the pointers, while the other implementation uses C bitfields to define the layout of the packed pointers.

Evaluation of different implementations

We use a synthetic, randomized benchmark run in userspace on Barrelfish to evaluate the different mapping database (MDB) implementations. Where

possible, we will show both a number of variations of the AA tree implementation and the DFS doubly-linked list implementation.

For each low-level operation, we perform 1000 measurements. We reset the mapping database after each measurement by filling a capability slot array of varying size (from 4096 up to 65536 slots) with naturally-aligned RAM capabilities. Measurements taken in Barrelfish indicate that a fully booted, idle system may have roughly 3000 capabilities, of which 99% are derived from PhysAddr. Roughly 20% of the capabilities have copies, 10% have ancestors, and 5% have descendants.

We benchmark the operations listed in table 4.1. For finding the immediate predecessor and successor, we always benchmark finding the successors. In addition to finding the immediate successor, we also benchmark iterating over 10 and 100 successors.

For our benchmarks we generate capabilities randomly with an arbitrary, but fixed, seed provided to the libc `rand` function via `srand`.

We only set the random seed at the start of each experiment to get some variation in the exact location of the target capability in the tree or list.

We generate capabilities such that approximately 10% of the capabilities are copies of a capability in the other 90%, i.e. roughly 20% of all capabilities have copies. To approximate a "regular" distribution of capabilities we generate capabilities that are not elected to be copies such that many small capabilities but only a few large capabilities are created. We generate those capabilities 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.

Operation	Description
Insert	Insert one capability into the mapping database
Remove	Remove one capability from the mapping database
Predecessor/Successor	Return a capability's immediate predecessor or successor in the mapping database's ordering, as defined in section 4.5.2
Copies check	Check if the mapping database contains other copies of the given capability
Descendants check	Check if the mapping database contains descendants of the given capability
Ancestors check	Check if the mapping database contains ancestors of the given capability
Address range query	Query the mapping database for a capability in, or covering, the given range, cf. section 4.5.3

Table 4.1: *The set of low-level mapping database operations* $\mathbb{O}$

For measuring insertion latency, we create a mapping database that contains $n-1$ capabilities, and then insert the last capability while measuring the latency of the insertion using the hardware timestamp counter. For the other operations – removing a capability, finding a capability's successor(s), checking a capability's relations, and the address range query – we create the mapping database with n capabilities and subsequently measure the operation's latency using the hardware timestamp counters.

We execute all benchmarks on a 2x10 Intel Xeon E5-2670 v2 clocked at 2.5 GHz, and the machine is rebooted after taking 1000 measurements for

each operation.

For each operation we record each measurement individually and compute the median, standard deviation, and 99th percentile with `numpy` in python.

To provide a single "score" number for each implementation, we also measure the relative frequencies of the different operations for Barrelfish's boot phase and a process management workload, and compute the weighted sum of medians for each implementation.

For each workload, we count the low-level operations for the duration of the synthetic implementation of the workload.

The first workload is the boot phase of the system. For this workload, we read the operation counters at the start of the execution of our synthetic workload generator application, as we run the workload generator on a freshly-booted system to ensure reproducible numbers. For the system's boot phase we get the operation frequencies shown in table 4.2

Operation	Count	Frequency
Insert one element	25524	28.0%
Remove one element	16094	17.7%
Predecessor/Successor	10040	11.0%
Copies check	16493	18.1%
Ancestors check	6434	7.1%
Descendants check	13992	15.4%
Address range query	2469	2.7%
Total	91046	100.0%

Table 4.2: *Mapping database operation counts and frequencies during boot phase*

We see that the most frequent operation in the system boot phase is insertion, which accounts for 28% of the mapping database operations. The next most

frequent operations are removal and checking for copies which account for 17.7% and 18.1% respectively. This can easily be explained by looking at how Barrelfish's memory server operates. The deletes can mostly be attributed to the fact that the memory server, when splitting up capabilities using retype, will delete the source capability after the retype operation completes. We further see that checking for descendants and looking up predecessors and successors is more frequent with 15.4% and 11% respectively, than checking for ancestors and explicit range queries which account for the last 7.1% and 2.7%.

The next workload we consider is a system with a lot of short-lived application processes. This workload is important, as process creation and cleanup are two of the most capability-operation heavy system operations in Barrelfish. For this workload, we get the operation frequencies shown in table 4.3.

Operation	Count	Frequency
Insert one element	19300	19.9%
Remove one element	18779	19.3%
Predecessor/Successor	7472	7.7%
Copies check	19219	19.8%
Ancestors check	12063	12.4%
Descendants check	17884	18.4%
Address range query	2425	2.5%
Total	97142	100.0%

Table 4.3: *Mapping database operation counts and frequencies in process management workload*

From table 4.3, we can see that insertions and deletions each account for almost 20% of the overall operation mix in a process management heavy

workload. Another 19.8% of operations are checks whether a capability has copies. These three operations together make up about 60% of the total operations in the process management workload. Explicit queries for a capability's predecessor or successor make up 7.7% of the workload and checking whether a capability has descendants accounts for another 18.4% of the operation mix in the process management workload. Finally, we see that in the process management workload, explicit address range queries are only about 2.5% of the total count of operations, with another 12.4% of the operations utilizing range queries for the ancestor checks.

We use these frequencies to calculate the score $S_{w,i}$ for each implementation i and workload w as

$$S_{w,i} = \sum_{o \in \mathbb{O}} F_{w,o} \cdot Q_{2i,o},$$

where $\mathbb{O}$ is the set of low-level mapping database operations as listed in table 4.1. We denote the frequency of an operation $o \in \mathbb{O}$ in workload w as $F_{w,o} \in [0, 1]$, and the operation's median (2^{nd} quartile) latency for implementation i as $Q_{2i,o}$.

By choosing the workload operation frequencies $F_{w,o}$ to be in the interval $[0, 1]$, the score can be thought of as a number expressing the median latency in cycles per mapping database operation for the given implementation and workload.

We will now present measurements for each implementation discussed in section 4.5, before analyzing the trade-offs between different AA tree optimization.

Unless stated otherwise, the latencies we present are for a mapping database which contains 4096 capabilities.

Doubly-linked list vs. AA tree

First, let us look at the operation latencies for the doubly-linked list again, this time comparing them to the AA tree latencies. In this comparison, we want to show that our choice of data structure does reasonably well against the data structure we aim to replace.

We expect the tree to outperform the linked-list for insertion, and the ancestors check, while we expect significantly worse latency for removal and somewhat worse latencies for iterating over elements.

We show the median latencies for both the augmented AA tree, and the doubly-linked list in figure 4.3, with whiskers indicating the standard deviation. We indicate the exact latency in cycles for each bar by text positioned inside or above the bar. For bars where the standard deviation does not fit the plot area, we additionally indicate the standard deviation as text on the bar.

True to our expectations, we see a significantly better insertion latency for the augmented AA tree, both for the median and the standard deviation. The reason for the wildly fluctuating insertion latency for the linked list is the fact that, depending on where we need to insert a capability for which we have no prior knowledge of relations, we have to traverse a significantly different fraction of the list.

Again, as expected, removing a capability is much more costly in the AA tree, as the operation's complexity goes from $O(1)$ in a doubly-linked list to $O(logn)$ in the AA tree. Additionally, we see that iterating over elements is rather more expensive in the tree, as we follow more than one pointer on average to find a node's successor.

We see the most significant win for the tree in the check for a capability's ancestors. In the linked list, the median latency for checking if a capability has ancestors is around 22500 cycles. In the augmented AA tree, where

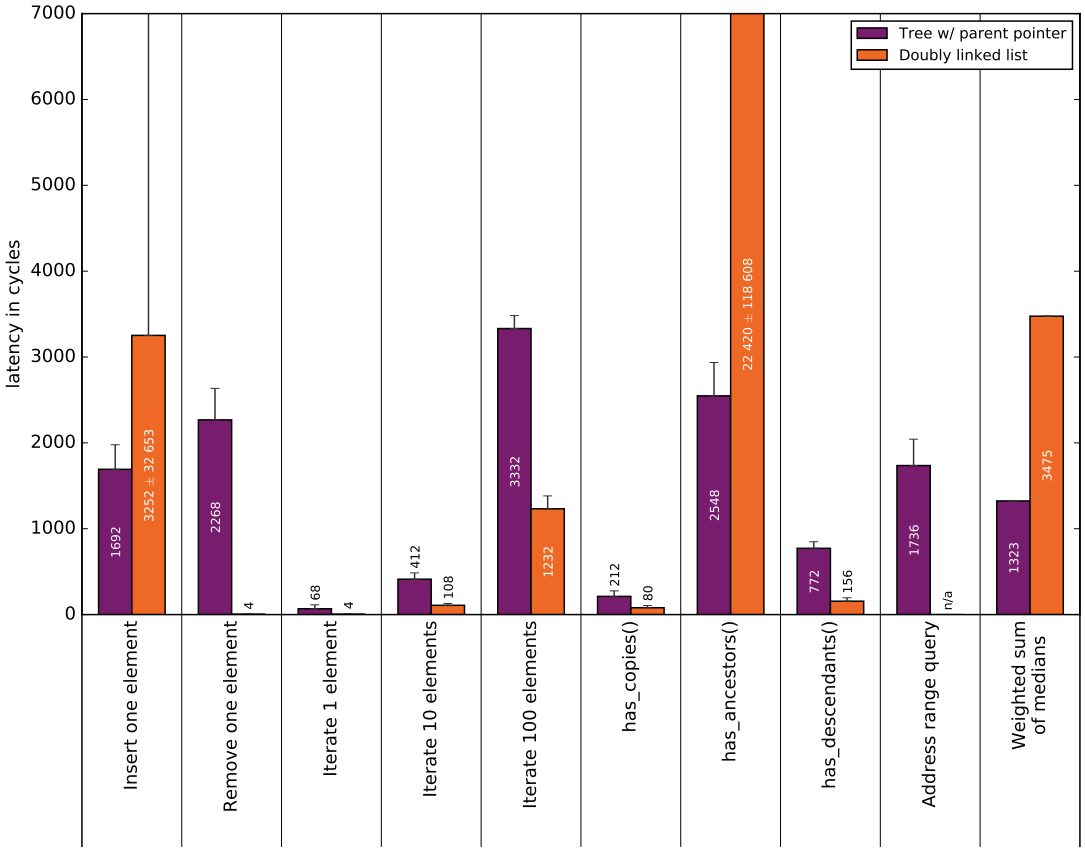


Figure 4.3: *Median latencies for low-level mapping database operations on a doubly-linked list and the augmented AA tree*

we implement the ancestor check using a range query, the ancestor check latency drops to only about 2500 cycles.

Overall, we see that the score for the process management workload for the doubly-linked list $S_{\text{procmgmt,linkedlist}}$ is 3475, whereas the score for the augmented AA tree is $S_{\text{procmgmt,tree}}$ is 1323. From these scores we conclude that choosing a binary search tree with support for range queries significantly improves mapping database performance for a process management heavy workload.

AA tree without a parent pointer

The next implementation that we discuss is the augmented AA tree without parent pointers which is one of the options that we propose for keeping capability size at 64 bytes.

This is the implementation that has been in use in Barrelfish since 2012 when we moved away from the linked-list implementation for the mapping database.

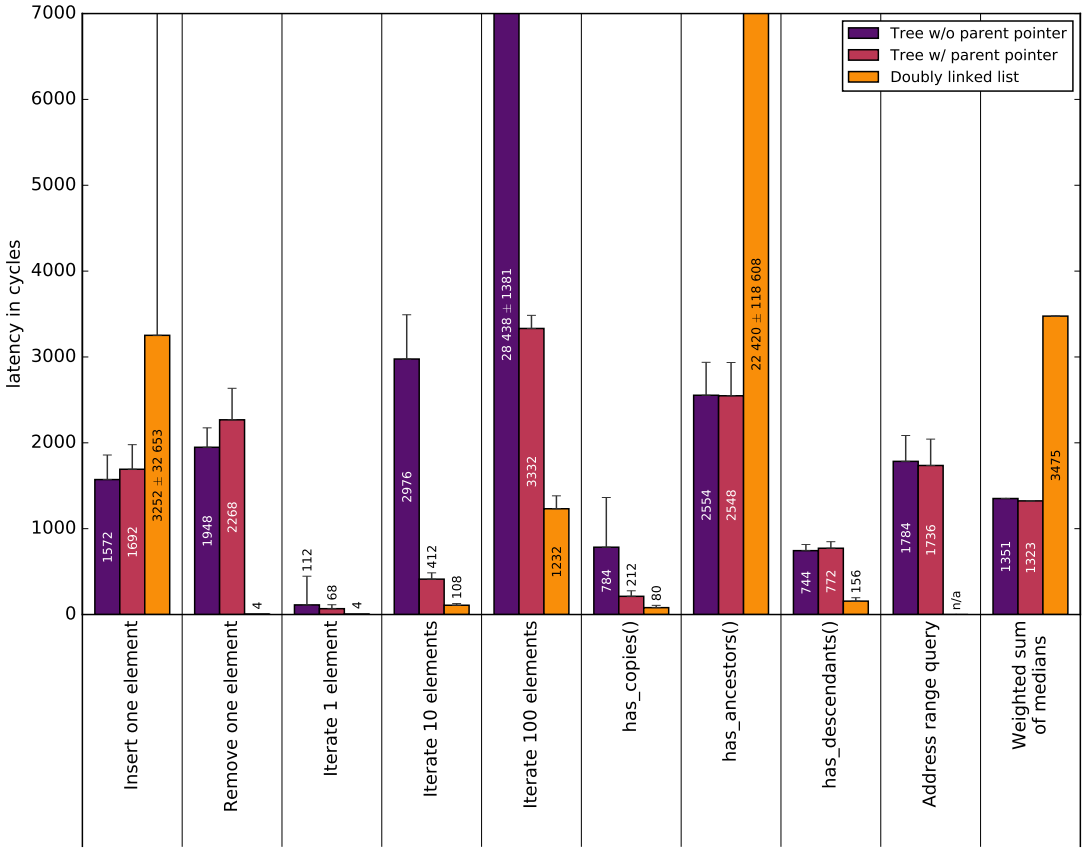


Figure 4.4: *Median latencies for linked list, tree and tree without parent pointers*

We see that most operations have comparable latencies for the tree with and without parent pointers. The lack of parent pointers is immediately

noticeable in the latency for finding immediate predecessors and successors, especially when we iterate over more than one element. For example, iterating over ten elements of the mapping database takes approximately 3000 cycles in the tree that lacks parent pointers, where the tree with parent pointers clocks in at around 400 cycles for iterating over ten elements. The check for a capability's copies is similarly affected because the check is implemented by looking at the capability's successor and predecessor and checking whether either of those two elements in the mapping database are copies of the capability we are checking.

Somewhat surprisingly, we see that insertion and removal have lower latencies the tree without parent pointers than in the regular tree. In fact, median insertion latency is 8% lower, and median removal latency is 15% lower in the tree without parent pointer. We believe this difference stems from the fact that inserting and removing nodes from a tree without a parent pointer needs only two pointer manipulations per modified node compared to three in a regular tree.

Overall, the score for the process management workload $S_{\text{procmgmt,tree-noparent}} = 1351$ is still significantly better than the doubly-linked list which is 3475.

Compared to the tree with parent pointers, we spend an extra 20 cycles per mapping database operation for the process management workload, which we can accept as a penalty to keep the capability size at 64 bytes.

AA tree with small data packed into pointers

We evaluate two techniques to implement packing small data items into the high bits of the AA tree pointers. The first technique relies on shifts and masking to extract and store the data values and pointers. The second technique uses C bitfields to get the compiler to extract and store the data values and pointers in the appropriate bits.

We expect to see latencies for the packed pointer implementations that are in the same ballpark as for the tree with parent pointers. However, we accept that the extra instructions necessary to pack and unpack the pointers and data items will lead to slightly increased latency.

Looking at the results in Figure 4.5, we can immediately discard the C bitfield variant, as it shows between 5 and 65% higher latencies than the shifts-and-masks variant.

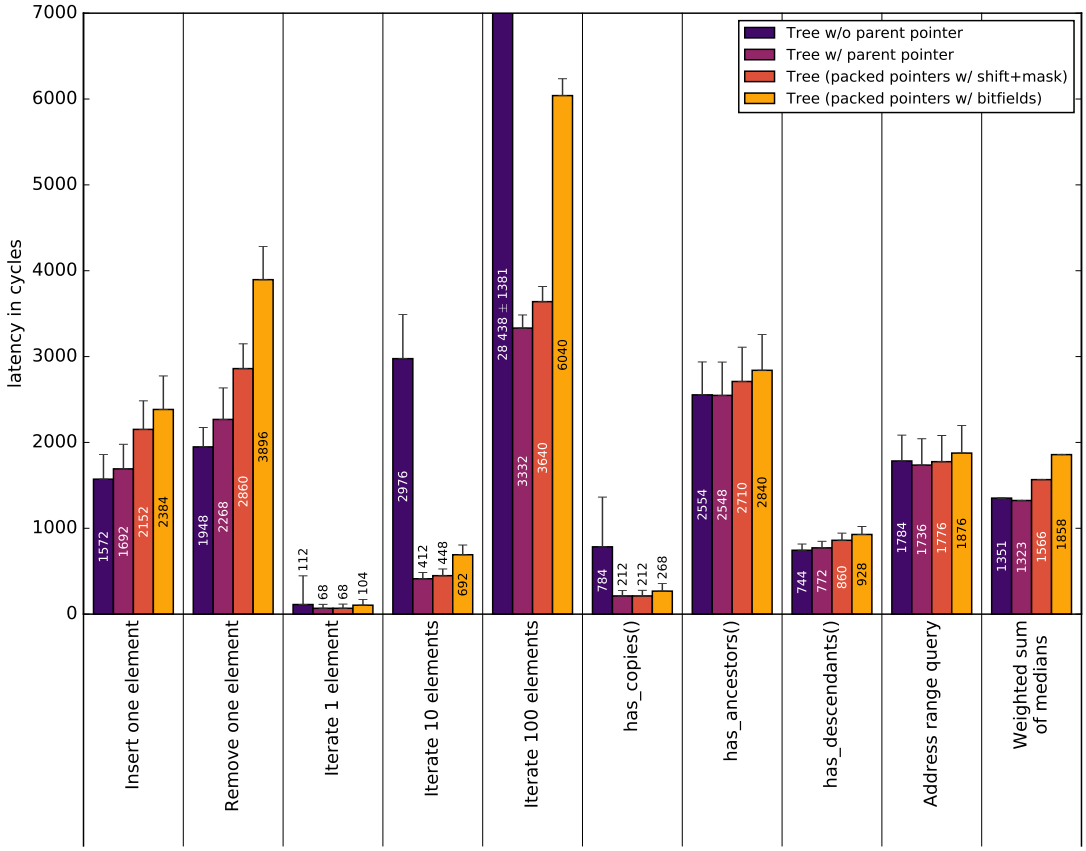


Figure 4.5: *Median latencies for tree, tree without parent pointers, and tree with packed pointers*

Looking at the shifts-and-masks variant in more detail, we see that insertion and removal show significantly higher latencies than the regular tree – 26% and 27% higher respectively. Additionally, we see a 8 to 9% increase in latency for iterating over 10 and 100 elements respectively when using packed

pointers. On the other hand, operations that do not require any parent pointer traversals, such as the range query, and the operations implemented in terms of a range query, are largely unaffected by the tree implementation choice regarding latency, and show a slowdown of 0% to check for copies, 4% to check for ancestors, 11% to check for descendants, and 2% for explicit range queries.

However, given the process management workload, both pointer packing variants show significantly worse scores $S_{\text{procmgmt,shift+mask}} = 1566$ and $S_{\text{procmgmt,bitfield}} = 1858$ than even the tree without parent pointers whose score is 1351.

We suspect that the large latency increases in insertion and removal stem from the fact that modern hardware does not have many execution units that are able to compute shifts and masks, which heavily impacts the amount instruction level parallelism we are able to achieve during tree rebalancing, as rebalancing requires a lot of pointer unpacking and packing. Further, because we utilize the high bits of the pointers to store data items, we may also impact the hardware's ability to prefetch tree nodes, as the pointers need to be unpacked before the hardware prefetcher can recognize the values as potential addresses.

Conclusion

To conclude, it is pretty clear that the augmented AA tree performs significantly better as the data structure of choice for the mapping database.

However, the question of choosing a tree implementation is significantly harder to answer. Given the process management workload we use for scoring here, the tree without parent pointers is the clear winner with a score that is only 20 cycles / operation worse than the regular tree's, while the two variants with packed pointers are clearly behind with scores of 1566

cycles/operation and 1858 cycles/operation for the shifts-and-masks and the bitfield packing respectively.

Depending on the workload however, a tree implementation with parent pointers may significantly increase application performance, which leads to the conclusion, that like so many other things in Barrelfish, the final choice on tree implementation may be better suited to a policy layer which could employ techniques like coreboot [ZGKR14] to provide applications with the best possible mapping database implementation given a predicted mix of mapping database operations resulting from the application's workload.

Finally, we show the median latencies for all the low-level operations for all implementations in figure 4.6.

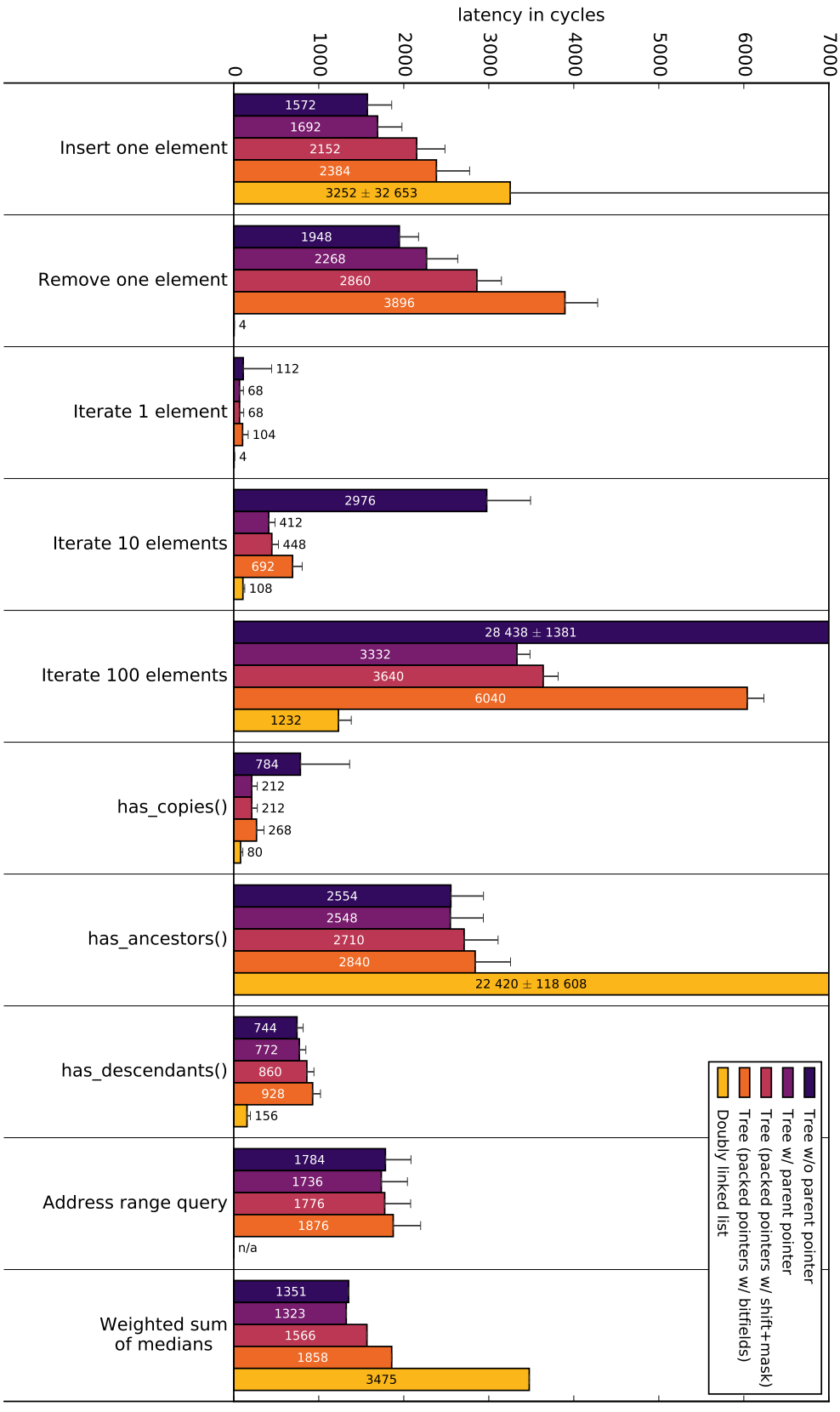


Figure 4.6: *Comparing different mapping database implementations for a mapping database containing 4096 capabilities. We show the 50 percentile with the standard deviation as error bars, except for the linked-list variant, where the error bars are astronomical. We give the latencies for operations in CPU cycles, and indicate the exact latency for each bar on or above the bar itself.*

Implementation in Barrelfish

In Barrelfish, we implement the protocol discussed in section 4.1 and section 4.2 in the *monitor*. Because the monitor is the user space component of the Barrelfish kernel, we have privileged access to all the capability spaces on the core. This is necessary in order to implement the capability operations. As the monitor is implemented in a purely event-based fashion, we implement all the capability operations based on events. Capability operations are initiated by applications sending a local RPC to the monitor. The monitor then executes the requested operation on behalf of the application. Each operation is implemented as a sequence of messages on the inter-monitor channels. Taken together, these messages move the capability on which the operation was invoked into a new state according to the state machine in figure 4.7.

We will take the DELETE operation as an example of how the algorithm presented in section 4.2 gets transformed to a sequence of kernel operations and messages between the monitors of a running Barrelfish instance.

We will first discuss the simple case, in terms of messages required, of deleting a capability which triggers further deletes. For our example we will assume that we are deleting a L2 CNode which holds some capabilities.

The delete is initiated by an application that tried to delete the CNode but got a return code of `SYS_ERR_RETRY_THROUGH_MONITOR` from the local CPU driver when calling `cap_delete` on the CNode's capability address. This return code triggers a capability delete RPC to the local monitor. The monitor receives the RPC call message and in turn calls the entry point for the DELETE operation in the monitor.

The first step in the monitor is to call `cap_delete` on the capability again, as its state could have changed according to figure 4.7 between the time

the application initiated the RPC and the monitor received the RPC. If this delete returns anything other than `SYS_ERR_RETRY_THROUGH_MONITOR` or `SYS_ERR_CAP_LOCKED`, we either got an actual error while deleting the capability, or the delete succeeded. In both those cases, we just forward the return code to the application, and the operation completes.

Otherwise, the monitor starts working through the DELETE algorithm. First the monitor sets up a state object which will track the state of this particular DELETE and fills out the object with all the necessary information, such as the kernel capability representation of the capability, the result handler callback and data, and some flags.

After that the monitor tries to lock the capability. This step is designed in such a way that it can be called repeatedly until the capability lock is acquired, or the capability has disappeared. Therefore, this step tries to delete the capability using the `cap_delete` call to the local CPU driver once again. If the simple delete returns `SYS_ERR_RETRY_THROUGH_MONITOR`, we can try to acquire the lock by calling requesting that the CPU driver locks the capability. In case the CPU driver returns `SYS_ERR_CAP_NOT_FOUND` for any invocations in this step, another operation, which was running concurrently, has deleted our target capability and we can signal success to the application. In case we get `SYS_ERR_CAP_LOCKED` from either of the invocations, we enqueue the function to be executed again once the capability is unlocked. This allows us to pause our operation without blocking the monitor until the capability is unlocked. This is important because some operations that lock capabilities can be long-running and we do not want to completely block the monitor until such an operation completes. Assuming the lock step succeeds, we have acquired the lock for the capability and can proceed with our DELETE. The first step after acquiring the lock is to check the capability's remote relations flags. This is an important optimization as we use these flags to decide whether we need to synchronize the delete across

cores.

From this point onward, there are three different scenarios to consider.

The first situation is that the capability we are deleting does not have remote copies. In this case we need to simply go through the process of properly deleting any capabilities contained in the capability we are deleting. This case happens most frequently when we are deleting CNodes.

If copies of the capability exist on other cores, there are two possibilities. If the capability has a type for which we can move ownership, we can simply transfer the capability's ownership to one of the cores that hold copies and then proceed to delete the now "foreign" copy. In the last case, where ownership cannot be moved, we need to first delete all copies on all other cores, after which we land in the first case described above, where no remote copies exist, and we clean up the very last copy of the capability that exists.

Cleaning up the last copy is a fairly complex process itself, as we have to potentially delete quite a few capabilities that were stored in the capability we are deleting.

The cleanup operation happens in two phases: first we inspect the capability and put all the capabilities contained in it on a list of capabilities that need to be deleted. If we find more capabilities which may contain further capabilities during this step, we also inspect those capabilities and put any further capabilities on the list, unless they are already there. This enables our algorithm to deal with circular references between CNodes in the manner we outlined in section 4.3.

This concludes the *marking* phase of the cleanup operation. This phase is currently implemented as a single invocation to the CPU driver, which may prove to be a problem when cleaning up a large CSpace. However, in day-to-day operation of Barrelfish we have not observed the latency of the mark invocation to be problematic.

Before starting the process of stepping through the list of capabilities to clean up, we enqueue our DELETE operation object to wait until the list is fully processed, that is, we have performed all the deletes and cleanups which are necessary for the original capability to be fully deleted. This process is called *delete stepping*, because it is implemented by processing the list of capabilities created in the mark phase one element at a time.

The delete stepping process is implemented as a queue of events in each monitor, and events on the queue are processed continuously as part of the monitor's event dispatching loop, unless the stepping mechanism is paused manually.

For each delete step, we invoke the capability on which the delete was requested. The metadata of this capability contains a pointer to the first element of the list of capabilities we still have to process. The CPU driver cleans up the capability at the head of the list and updates the head pointer to point to the next element of the list. Each delete step may produce a new RAM capability which needs to be returned to the memory server. If this is the case, the CPU driver puts the new capability into an empty slot which is specified as an invocation argument by the monitor. After each delete step, the monitor checks if the step produced a new RAM capability and forwards it to the memory server, if necessary.

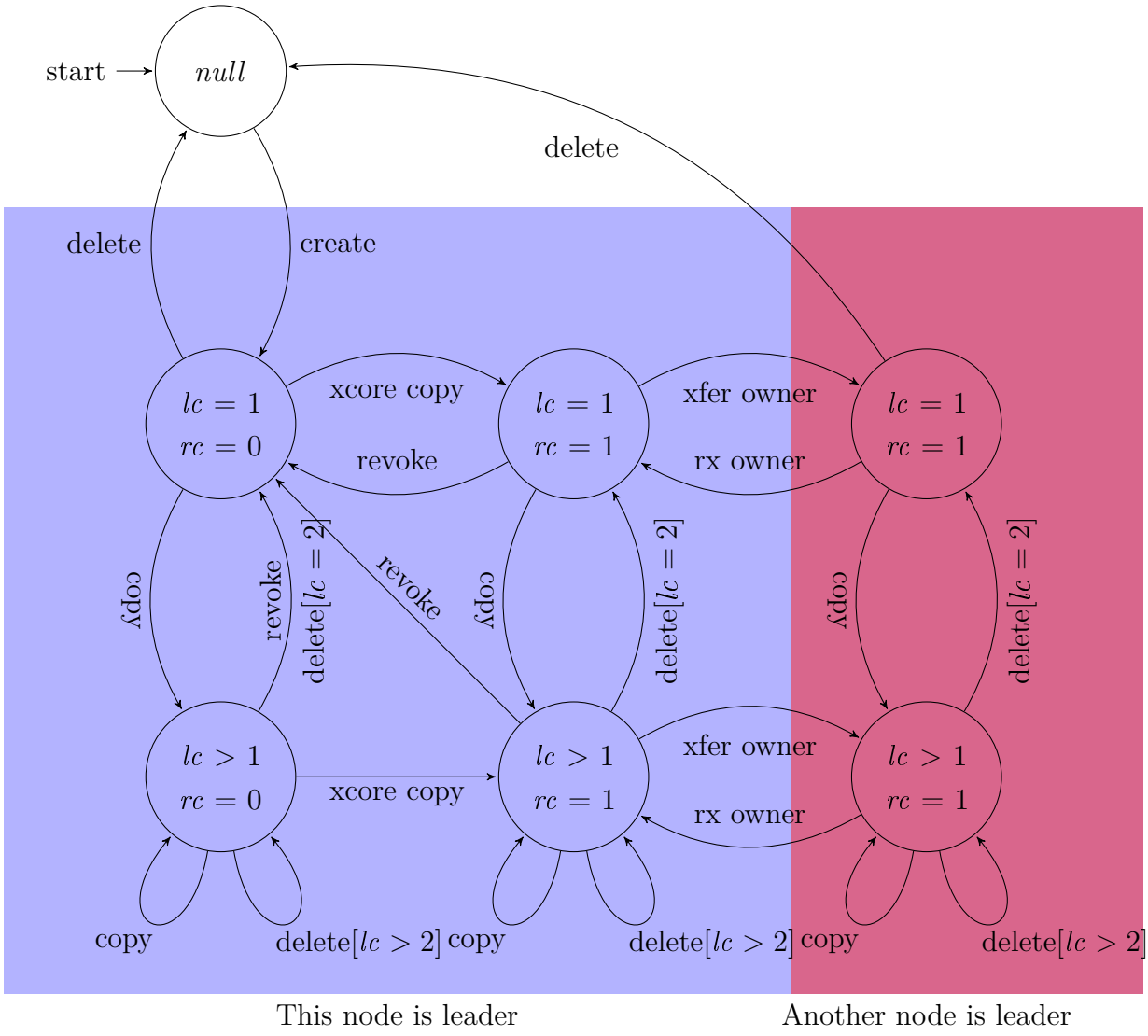


Figure 4.7: *The state machine for a single capability slot. This is the state machine which is implemented in Barrelfish for the* Delete, Chown, *and* Revoke *operations.*

Evaluation

In this section we will discuss the experimental evaluation of the distributed capability system presented in this chapter. We have designed microbenchmarks which cover all the capability operations discussed earlier in the chapter. The basic format of these microbenchmarks is that we measure the latency of a capability operation while varying the number of capabilities in the local mapping database.

With these microbenchmarks we want to verify that the capability operation implementations fulfil our requirement of not showing unreasonably high latency. As most operations will include operations on the mapping database, we expect to see logarithmic latency behaviour with respect to the total amount of capabilities present on a core, i.e. the mapping database size on that core.

Furthermore, we present a series of latency breakdowns with which we analyze the latency of the different parts of each operation. Using these breakdowns, we will also identify parts of the implementation which need further attention.

Experimental design

Again, we conduct all experiments on a 2x10 Intel Xeon E5-2670 v2, clocked at 2.5 GHz. More details about the machine are given in table 2.2.

For the latency breakdowns, we work with a fixed number of capabilities present in the mapping database. We choose to populate the mapping database with 4096 extra capabilities over the number of capabilities that exist in the mapping database of an idle core in the system . We make this choice, as we observe that typical workloads usually result in similar mapping

database sizes. We use Barrelfish's built in tracing infrastructure [SG13] to capture timestamps at key points inside the operation implementations. The evaluation of the tracing infrastructure [SG13, §5.3.1] shows that recording a trace point has a latency of roughly 40 cycles. Considering this, and taking into account the fact that we utilize less than ten trace points for each operation that we benchmark, we accept that the tracing infrastructure may increase the latency of a purely local operation, such as deleting a local copy of a capability, by roughly 10%. For operations that require synchronization, and have much higher latency, the latency contributed by the tracing infrastructure is less than 0.2%.

Invoke

We look at the INVOKE operation first, as all the other operations are implemented as invocations on CNodes. Apart from evaluating invoke itself, we will use this experiment to establish how much latency an invocation itself contributes to the latency of the other capability operations.

Latency of a "noop" invocation Because invocation targets are given by their capability address we do not expect the mapping database size to influence invocation latency. To illustrate this, we present the latency for a "noop" invocation here. As we can see in figure 4.8, the cost of transferring control to the CPU driver and back via a `SYS_INVOKE` system call is mostly less than 250 cycles. As predicted, the number of capabilities in the mapping database does not impact invocation latency, because resolving a capability by its capability address is a constant time operation, cf. the discussion about capability addresses in section 2.6.

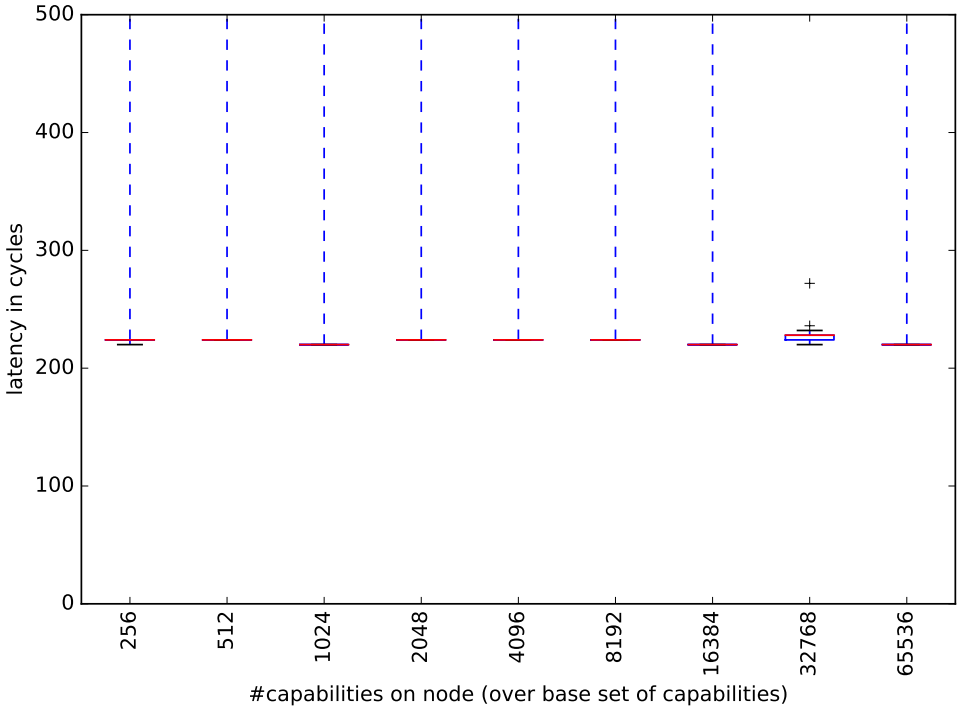


Figure 4.8: *"noop" invocation latency*

Delete

Next we present a series of experiments showing the latency for the DELETE operation while varying the number of capabilities in the mapping database. We consider the following states in which a capability may be when it is deleted:

1. The capability is *local*, and other *local* copies exist.
2. The capability is *foreign*.
3. The capability is *local*, and no other copies exist.
4. The capability is *local*, and *foreign* copies exist.

We expect that deleting a capability in each of the four states will have different absolute latencies, but states 1 and 2 should exhibit similar trends.

Deleting a local capability with local copies The setup for this experiment is quite simple. We just allocate a RAM capability on any core, create a copy of it, and measure the latency of deleting that copy.

Because the only operation that is not $O(1)$ when deleting a local capability with local copies is removing a node from the mapping database, cf. algorithm 3, we expect to see relatively low latencies absolutely speaking, and a logarithmic increase in latency, as removing a node from the mapping database is $O(\log n)$ on average with n being the number of capabilities in the mapping database.

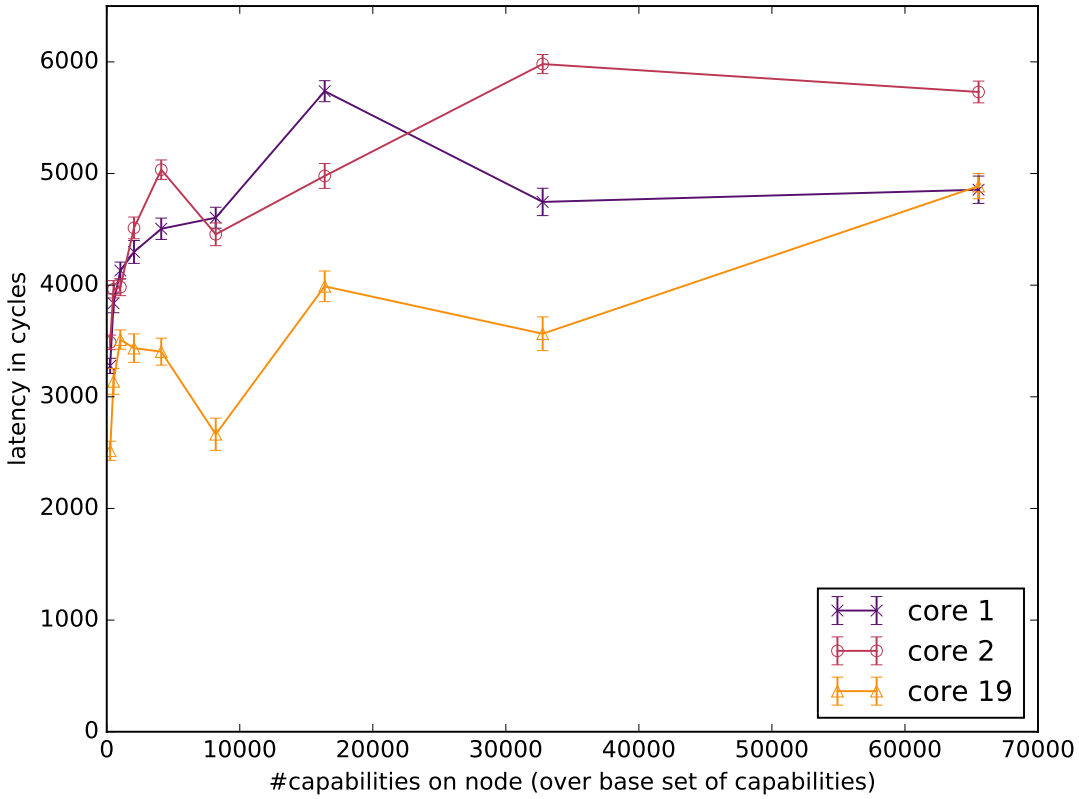


Figure 4.9: *Deleting a* local *capability which has* local *copies*

Figure 4.9 shows that the latency for deleting a local copy is between 2000 cycles and 6000 cycles depending on the amount of capabilities that exist on the core, with a clearly logarithmic trend.

Looking at a more detailed breakdown of the delete latency, we see that more than half of the latency, 2428 cycles of 4092 cycles for a mapping database with 4096 synthetically added capabilities, is contributed by the call to `mdb_remove`, which is the C function for removing a capability from the mapping database. Another 1000 cycles are spent in the check whether the capability we are deleting has local copies. The latency for this check is rather high, as we choose the mapping database implementation that uses the AA tree without parent pointers as the default implementation, cf. subsection 4.5.4. Interestingly, in our breakdown, the system call latency is more than 500 cycles, increasing by a factor of two compared to the no-op invocation which shows latencies under 250 cycles. This increase in latency can be contributed to two factors. First, as delete, like all CSpace manipulations, is implemented as an invocation on a root CNode, we need to do an extra capability lookup to find the capability which we want to delete. Second, as we use Barrelfish's tracing framework to acquire the latency numbers necessary to produce the breakdown, we introduce a number of operations – time stamp counter reads etc. – which simply are not present in the benchmark execution where we only report the overall delete latency.

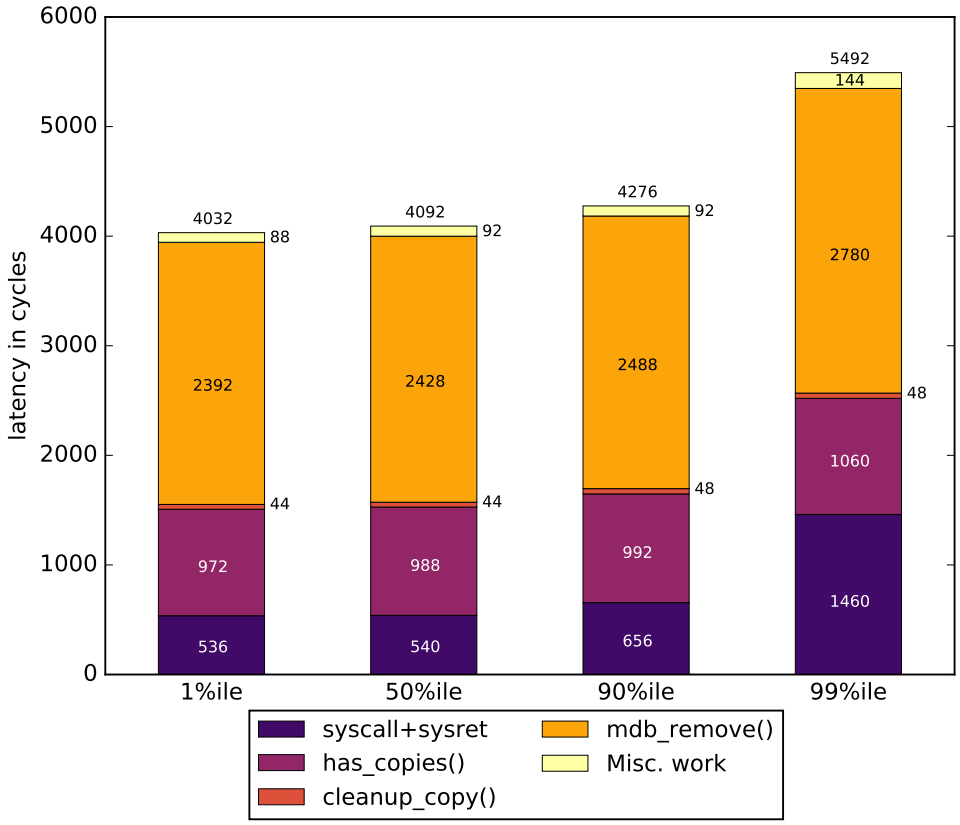


Figure 4.10: *Latency breakdown for deleting a* local *capability with local copies*

Deleting a foreign capability In this experiment we create a foreign capability by first allocating a RAM capability on one core and then transferring a copy of that capability to another core without moving the capability's ownership. We then repeatedly create a copy and delete that copy on the core that is not the capability's owner.

Similar to deleting a local capability for which local copies exist, the only mapping database operation necessary for deleting a foreign capability is the removal of the node in the tree. Thus we again expect the latency to be logarithmic, but still low absolutely speaking.

In this plot, figure 4.11, we see that the latency of deleting a foreign capability copy is between 6000 cycles and 10 000 cycles for mapping databases with

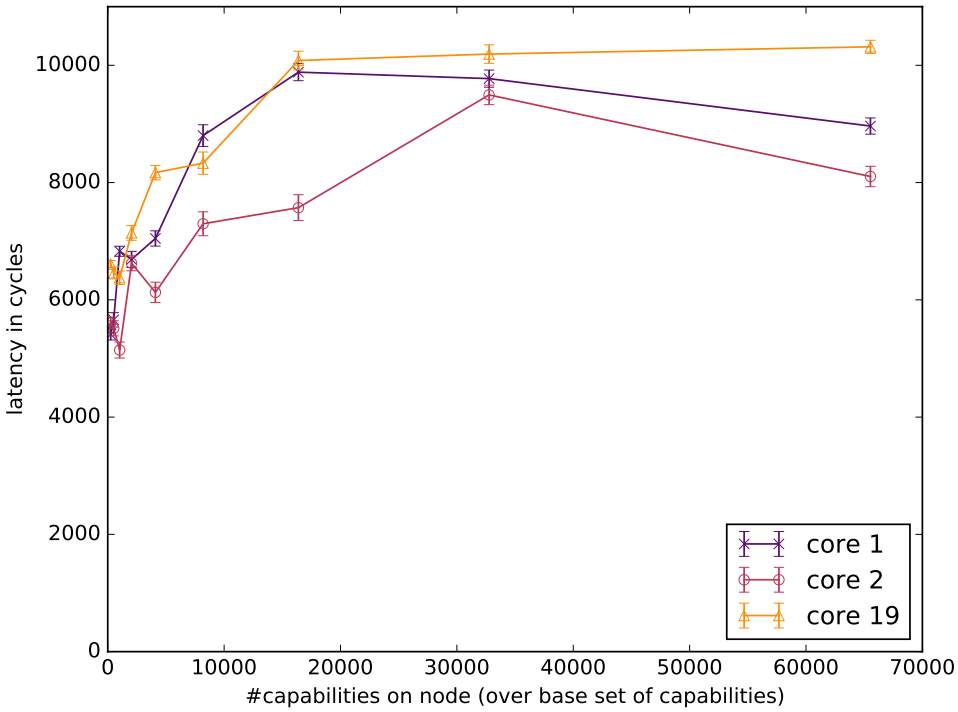


Figure 4.11: *Deleting a* foreign *capability*

about 1000 to about 65000 synthetically added capabilities.

This is surprisingly higher than the latency for deleting a local capability, so we investigate in more detail by looking at the latency breakdown for deleting a foreign copy shown in figure 4.12.

We see that in contrast to deleting a local capability, the latency of the call to `cleanup_copy` contributes 2476 cycles to the total latency, which accounts for the roughly 2000 cycles higher latency compared to deleting a local copy. The reason why the latency of `cleanup_copy()` is much higher when deleting a foreign copy is that we need to ensure that the deleted capability's ancestor's remote descendants flag is set, if an ancestor exists on this core. Finding such an ancestor requires a range query which, as discussed in subsection 4.5.3, has an asymptotic complexity of $O(\log n)$ on average.

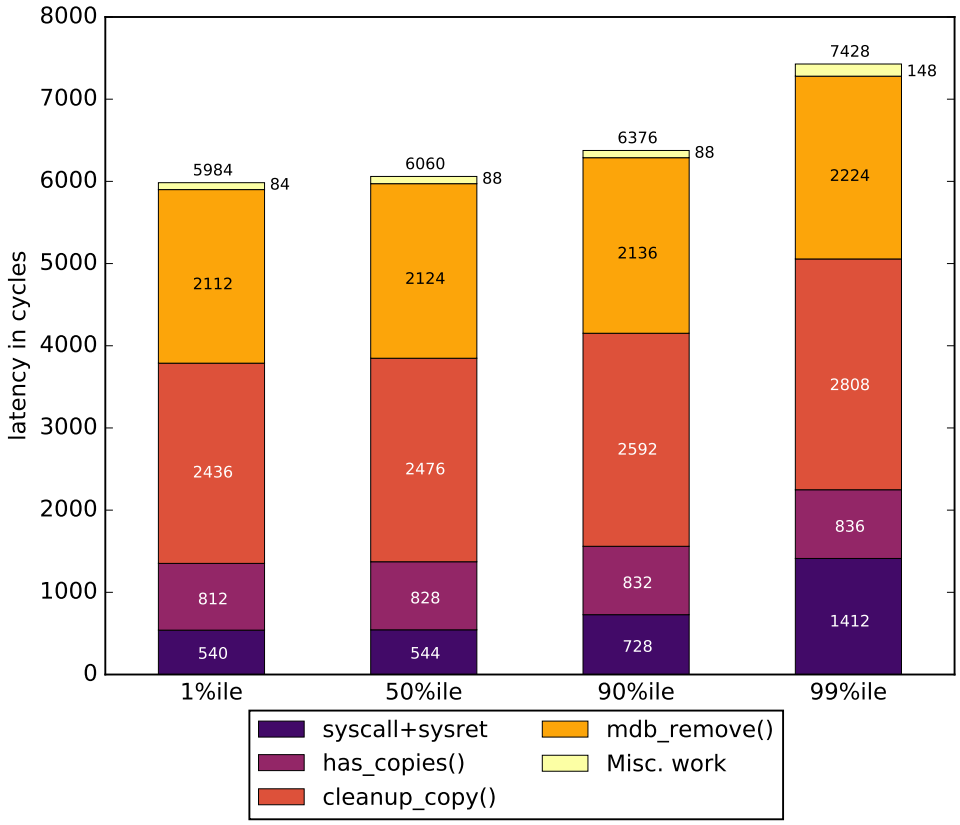


Figure 4.12: *Latency breakdown for deleting a* foreign *capability*

Deleting a local capability with foreign copies For this experiment we need to make sure that the capability we delete is the last local copy. To achieve this, we allocate a new RAM capability for each benchmark iteration. We then distribute copies of that RAM capability to other cores in the system. After the copies have been distributed, we measure the latency of calling `cap_delete` on the capability which we originally allocated. As a last step of each benchmark iteration, we delete all the remaining copies we distributed in order to keep the mapping database size constant across iterations.

In this experiment, we measure the latency of Delete when, in addition to removing a tree node, the system has to find another core in the system that has *foreign* copies and make that core the new owner. Thus we expect

to see higher absolute latencies, as we need to send messages to other cores in the system.

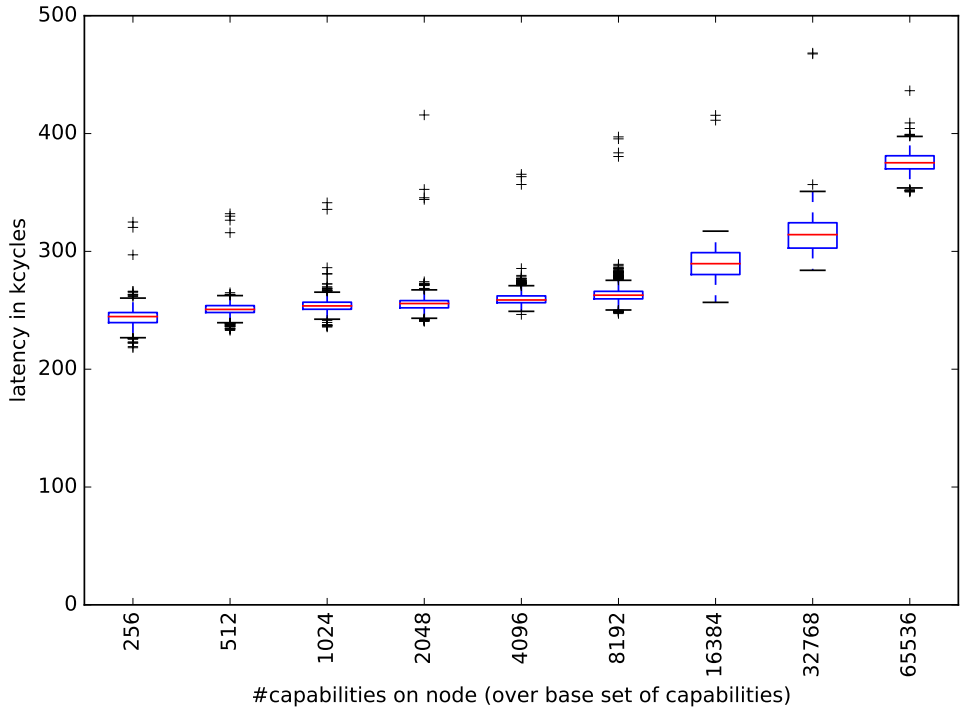


Figure 4.13: *Deleting a* local *capability which only has* foreign *copies*

As we can see in figure 4.13, this case has latencies which are about two orders of magnitude larger than the latencies observed when we do not have to communicate with other cores. We do not show 130 latency outliers above 500 kcycles across all mapping database sizes, as those are mostly artifacts which stem from scheduling decisions.

Given the latency breakdown shown in figure 4.14, we can see that lion's share of the latency for this operation, approximately 215 kcycles of the median latency of 257 kcycles given a mapping database with 4096 synthetically generated capabilities, is contributed by the broadcast to find a new owner and the follow-up RPC to move the capability's ownership to that core.

The remaining 42.7 kcycles are made up of the RPC between the benchmark application and the monitor on the local core, and some local work that

needs to be done before and after finding a new owning core.

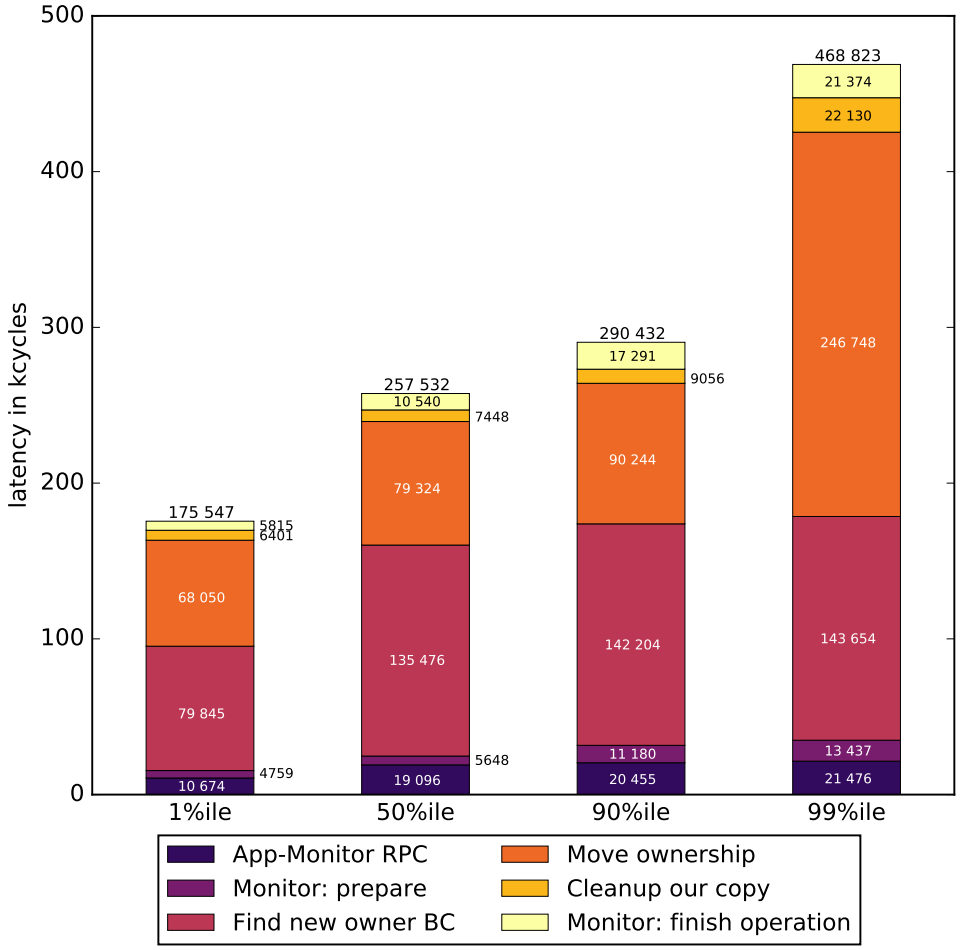


Figure 4.14: *Latency breakdown for deleting a* local *capability with only* foreign *copies*

Deleting a local copy for which no other copies exist The last case of deleting a local capability which we consider separately is the case where we delete the last capability that refers to some resource (usually RAM). This case is interesting, as apart from deleting the capability, the system needs to ensure that no physical resources are leaked. When the last capability referring to some region of RAM is deleted, Barrelfish will create a new RAM capability covering the region. That new capability then

needs to be passed to the memory server. This is achieved by first passing the capability to the local monitor which can forward the capability to the correct memory server.

To measure the latency of delete in this case, we repeatedly allocate a new RAM capability and measure the latency of doing a DELETE on that capability.

As an optimization, the CPU driver tries to process such a delete without the application having to do an explicit user level RPC to the monitor. The new RAM capability is passed to the monitor by simply inserting it into the well-known monitor upcall endpoint that is available to the CPU driver. However, if there is no space left in the monitor upcall endpoint, the CPU driver will signal the application that it has to retry the delete that would result in a new RAM capability through the monitor.

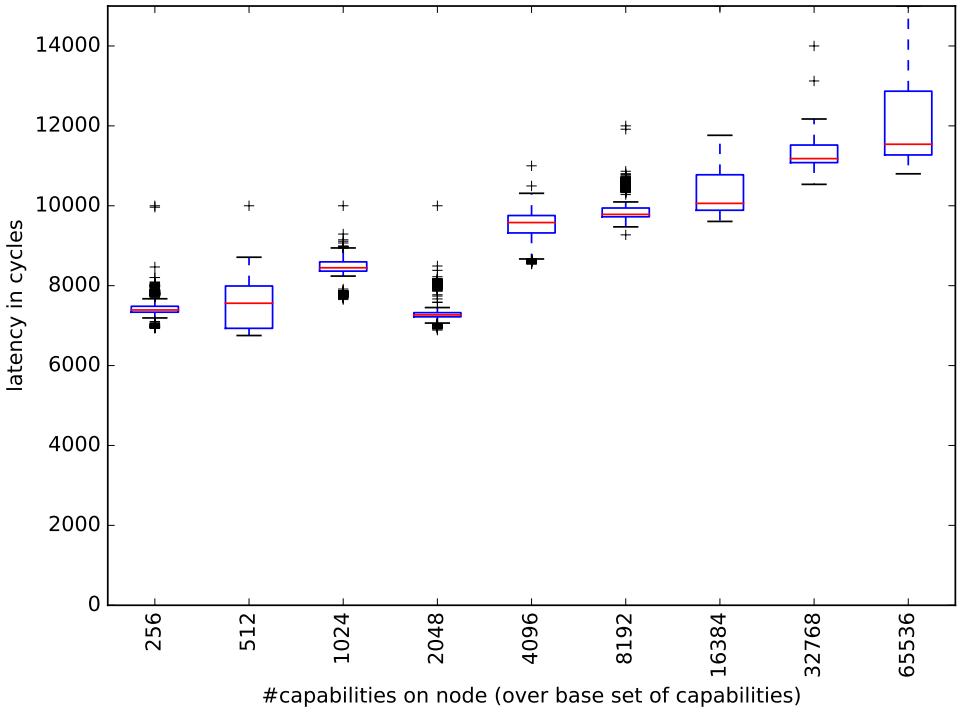


Figure 4.15: *Deleting last copy of a (*local*) capability*

As we can see in figure 4.15, this case of DELETE is a fairly low-latency op-

eration, taking between 6000 cycles and 15 000 cycles for mapping database sizes up to approximately 65000.

Looking at the latency breakdown for this delete variation, figure 4.16, we see a call to `cleanup_last` which is responsible for creating a new capability if necessary. This call contributes roughly 40%, 4356 cycles, to the total median latency of 10 168 cycles for a mapping database with 4096 synthetic capabilities. The other two operations that make up 24%, 2500 cycles, and 19%, 1936 cycles, respectively, are removing the capability from the mapping database, and checking whether copies exist for the capability.

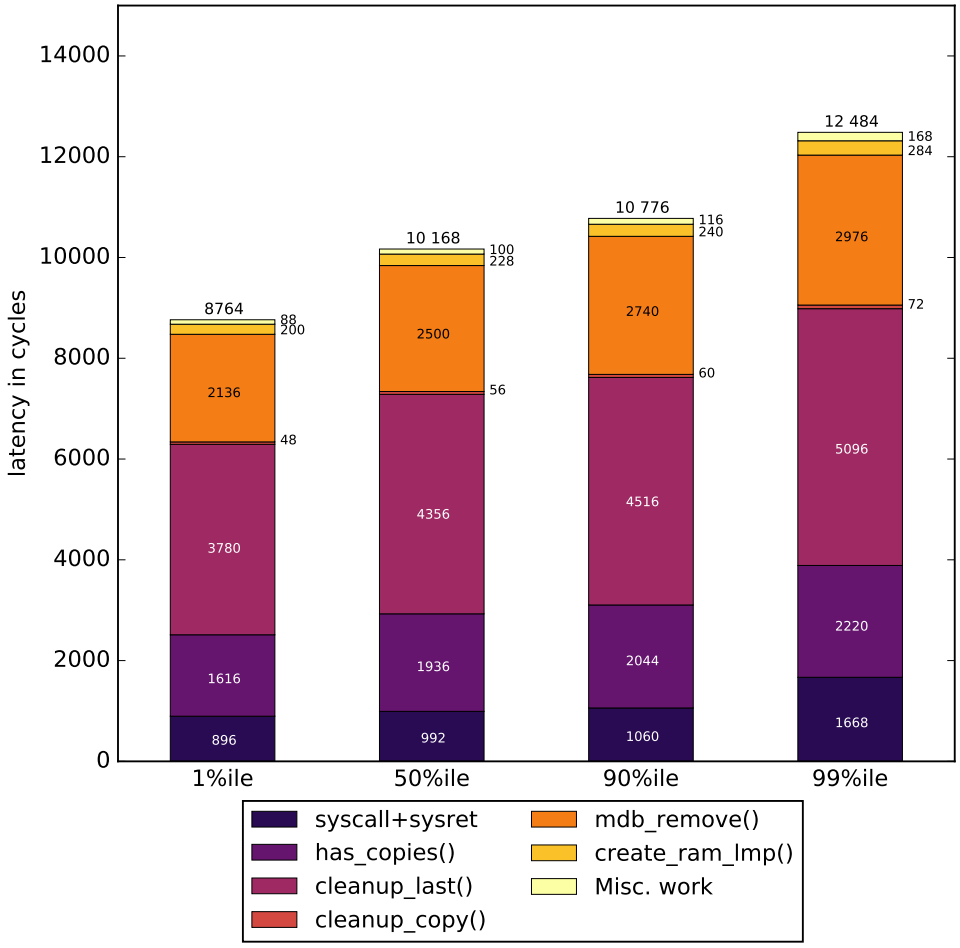


Figure 4.16: *Latency breakdown for deleting the last copy of a (*local*) capability*

This – again – shows that deletes that do not involve other cores and thus can be processed in a single invocation are fast operations.

Deleting CNodes

Deleting a CNode is a special case, as we have to delete the CNode's contents when deleting the last copy of a CNode capability. We discuss the process in more detail in section 4.6. Here we show two different microbenchmarks, one is how the latency of deleting a CNode behaves when we vary the mapping database size, and the other is how the delete latency changes depending on the number of slots occupied.

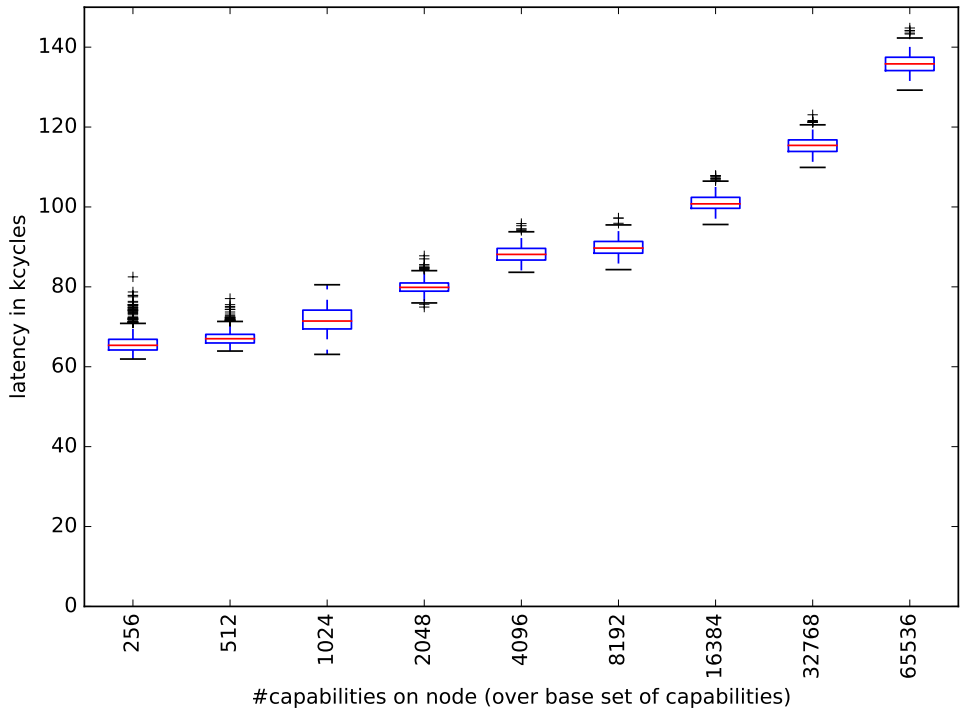


Figure 4.17: *Deleting last copy of a CNode with 4 occupied slots*

For the first benchmark, where we evaluate the latency of deleting a CNode while varying the number of capabilities on the core, we create a new CNode in each iteration and populate the first four slots of the CNode with RAM

capabilities that are allocated from the memory server. This requires a fair amount of work, as we ensure that the RAM capabilities which are present in the CNode that is deleted refer to regions for which no other capabilities exist. This choice tries to model the most frequent case of deleting a CNode: deleting an application's CSpace after the application exits. In that case we expect that a some amount of capabilities present in the CNodes will be capabilities referring to regions for which no other capabilities exist. The CNode delete will create new RAM capabilities for any regions of physical memory for which the last capability is deleted, as discussed in section 4.3 and evaluated in isolation in the previous experiment.

Figure 4.17 shows the results of this benchmark for mapping database sizes with 256 up to 65536 synthetically added capabilities. For a CNode in which four slots hold a RAM capability that needs to be returned to the memory server we see latencies from 65 kcycles up to 140 kcycles for differing mapping database sizes.

Qualitatively, we see that deleting a CNode has a behavior similar to deleting any other memory based capability. However, the absolute latency is an order of magnitude higher, because we require a system call for each *delete step*, as discussed in section 4.3.

In figure 4.18, we see that processing the actual delete takes approximately 48 kcycles of an overall operation latency of 80 kcycles. This is in the right ballpark, as we have shown in figure 4.15 that the latency for a single cleanup operation is approximately 10 kcycles.

For the second benchmark, we do not create any extra capabilities on the node running the benchmark, but rather vary the number of occupied slots in the deleted CNode. Again, we choose to a allocate RAM capability referring to a region for which no other capabilities exist for each CNode slot that we wish to fill. This experiment tries to show the impact of the

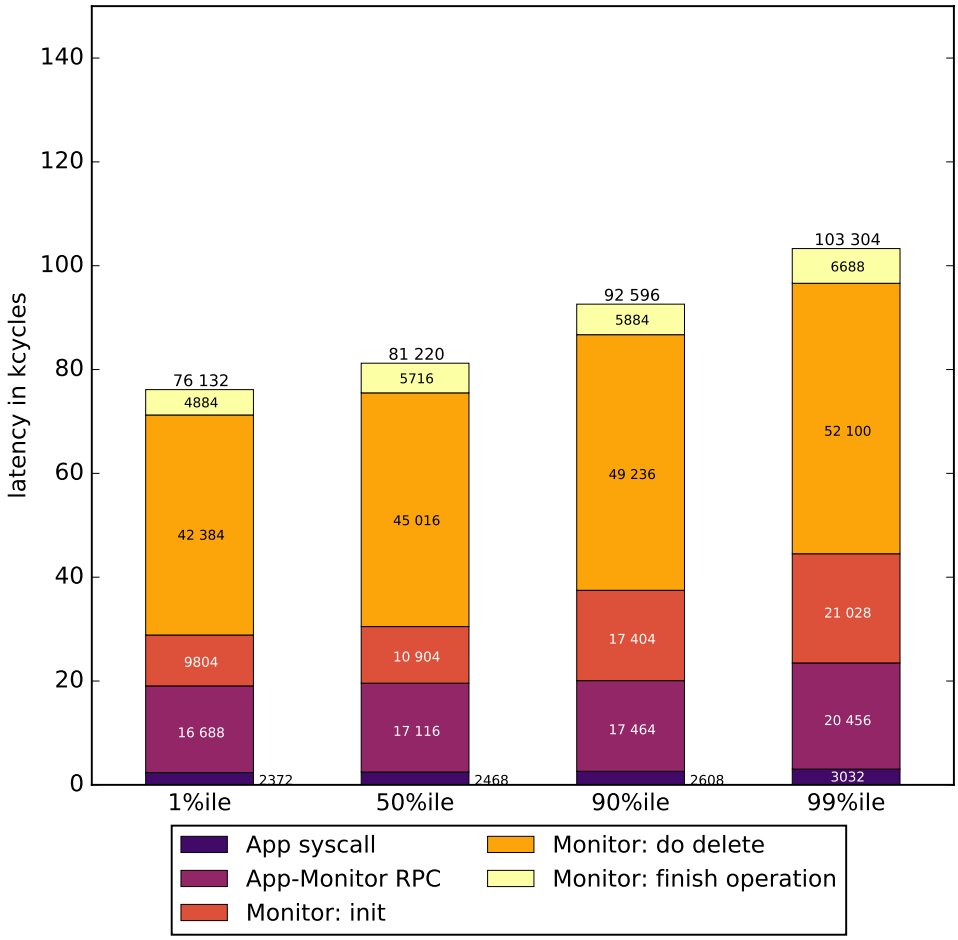


Figure 4.18: *Latency breakdown for deleting the last copy of a CNode with 4 occupied slots*

delete steps, as we need one delete step per slot in the CNode that holds a capability that needs to be returned to the memory server.

Figure 4.19 shows the latency for deleting a CNode which has 1 to 256 slots occupied by capabilities that need to be cleaned up. We see that while deleting a cnode with only a couple slots occupied has fairly low latency, approximately 50 kcycles, a fully occupied CNode takes approximately 1.7 Mcycles to delete.

Given that we need one delete step per occupied slot in this experimental setup, it is unsurprising that the latency has a linear response to the number

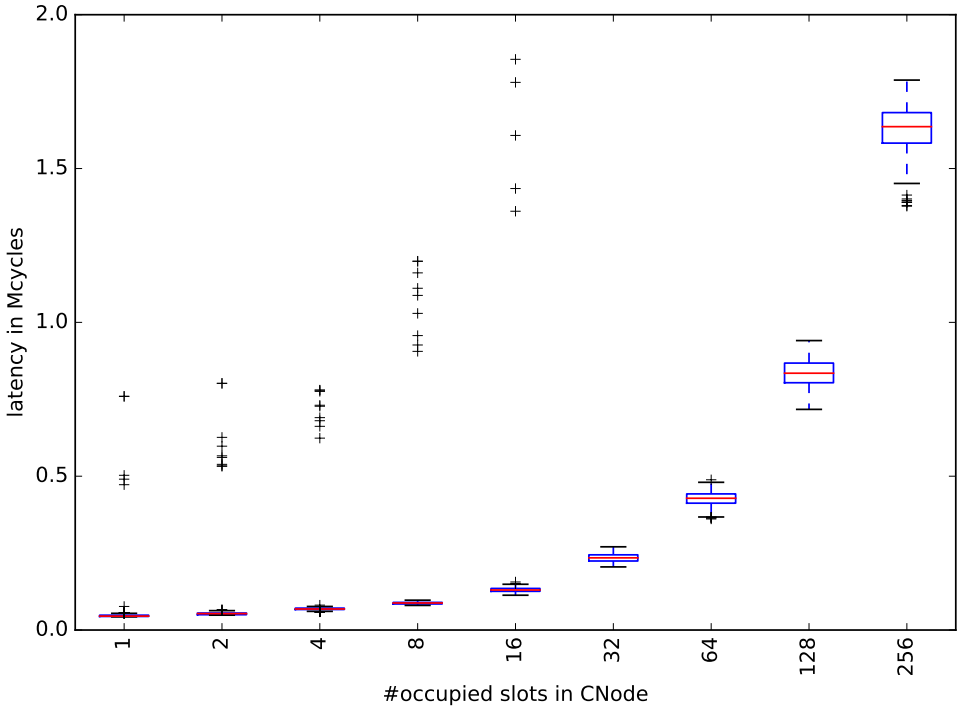


Figure 4.19: *Deleting last copy of a CNode while varying number of occupied slots*

of occupied slots.

Using the numbers from the breakdown plot in figure 4.18, we see that in the monitor we are more efficient at deleting and reclaiming memory capabilities, as we would predict a latency of $256 \times 10\,\text{kcycles} = 2.56\,\text{Mcycles}$ for the delete steps that are required to delete a fully occupied CNode. This prediction is approximately 50% higher than the latency of 1.7 Mcycles that we actually observe in this experiment.

Revoke

Now that we have gained an understanding of the latencies for different cases of DELETE, we consider the following cases for REVOKE:

1. Revoking a capability with no *foreign* relations
2. Revoking a *foreign* copy of a capability
3. Revoking a *local* copy of a capability with *foreign* relations

It is important to remember that a REVOKE is simply a series of deletes that should appear atomic to callers.

For each benchmark, we create 10 copies of the capability which we will revoke. Again, we measure the latency of 1000 calls to revoke for each mapping database size.

Revoking a capability with no foreign relations First, we present the latency for revoking a capability which has no foreign relations. In this case, for each mapping database size, we allocate a RAM capability from the memory server. Before each call to revoke, we create ten copies of the capability, which will be deleted by the revoke operation.

Given that the bulk of the work for this case of revoke is deleting the existing ten copies, we predict that the revoke latency is going to be comparable to ten times the latency for deleting a local capability for which local copies exist.

$$Latency_{\text{revoke w/o remote}} = 10 \times Latency_{\text{delete local copy}}$$

Referring to the benchmark deleting a local copy of a capability for which other local copies exist in section 4.7.3, we see that the median latency for

deleting a local copy on a node with a mapping database that contains 4096 synthetic capabilities is roughly 4100 cycles.

$$Latency_{\text{delete local copy}} \approx 4100\,\text{cycles}$$

This gives a rough estimate for the median latency of revoking a local capability with ten local copies, and no remote relations on a node with a mapping database that contains 4096 synthetic capabilities

$$Latency_{\text{revoke w/o remote}} \approx 41\,000\,\text{cycles}$$

We show the latency for revoking a capability with no foreign relations in figure 4.20.

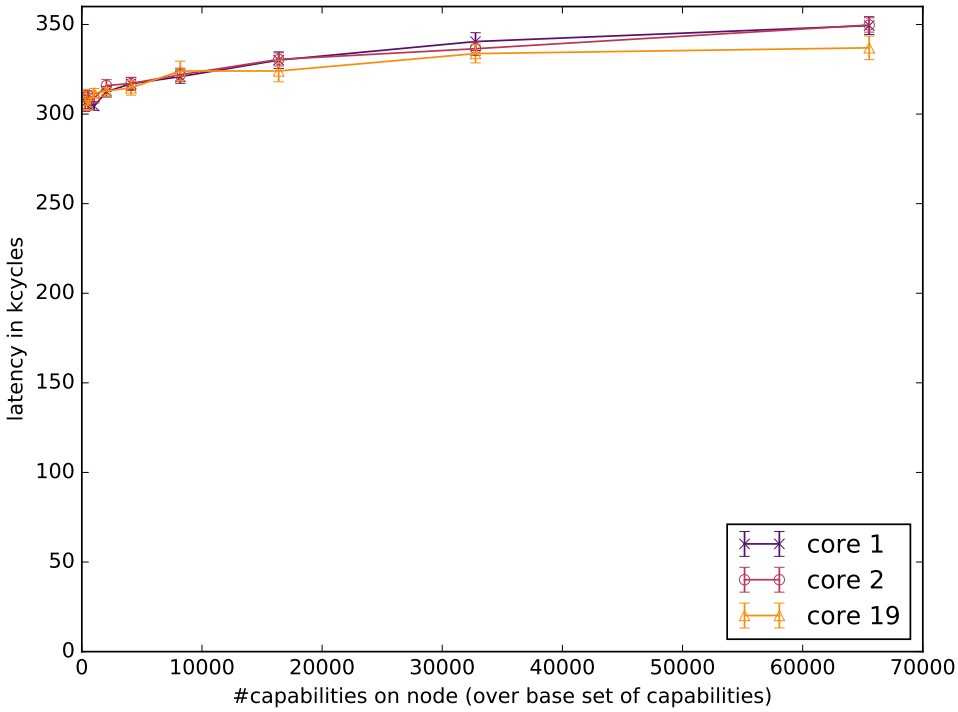


Figure 4.20: *Revoking a capability with no* foreign *relations*

We see that the latency for revoke in this case is between 300 kcycles and 350 kcycles, with no outliers omitted. The latency for revoke is much higher than our estimate, because we do a full two-phase commit on all the nodes

in the system for every revoke, even if there exist no remote relations for a capability. This is necessary, because the cached indicators that the CPU driver uses to tell whether a capability has remote relations may be stale. Additionally, we require the full operation for every revoke, so we can ensure that we never miss a concurrent copy operation that may be in progress when the revoke is requested.

Looking at the latency breakdown, figure 4.21, we see that, indeed, the most of the overall latency comes from the two system-wide broadcasts which form the two phases of our commit. Each phase has a median latency of about 135 kcycles, independently of the number of capabilities present in the mapping databases on all nodes.

For remaining 65 kcycles, the RPC between the benchmark application and the monitor contributes approximately 16 kcycles. The actual deletions happen in the monitor "prepare" phase which has a median latency of 41 716 cycles given a mapping database that contains 4096 synthetic capabilities. The latency for the "prepare" phase almost perfectly matches our predicted latency $Latency_{\text{revoke w/o remote}}$, showing that the latency for the actual work that needs to be done can be approximated fairly well by simply multiplying the latency for deleting a local copy by the number of capabilities that need to be deleted for the revoke.

The remaining cycles, less than 10 kcycles, are contributed by various bits of monitor code, such as cleaning up temporary capability copies after the revoke completes.

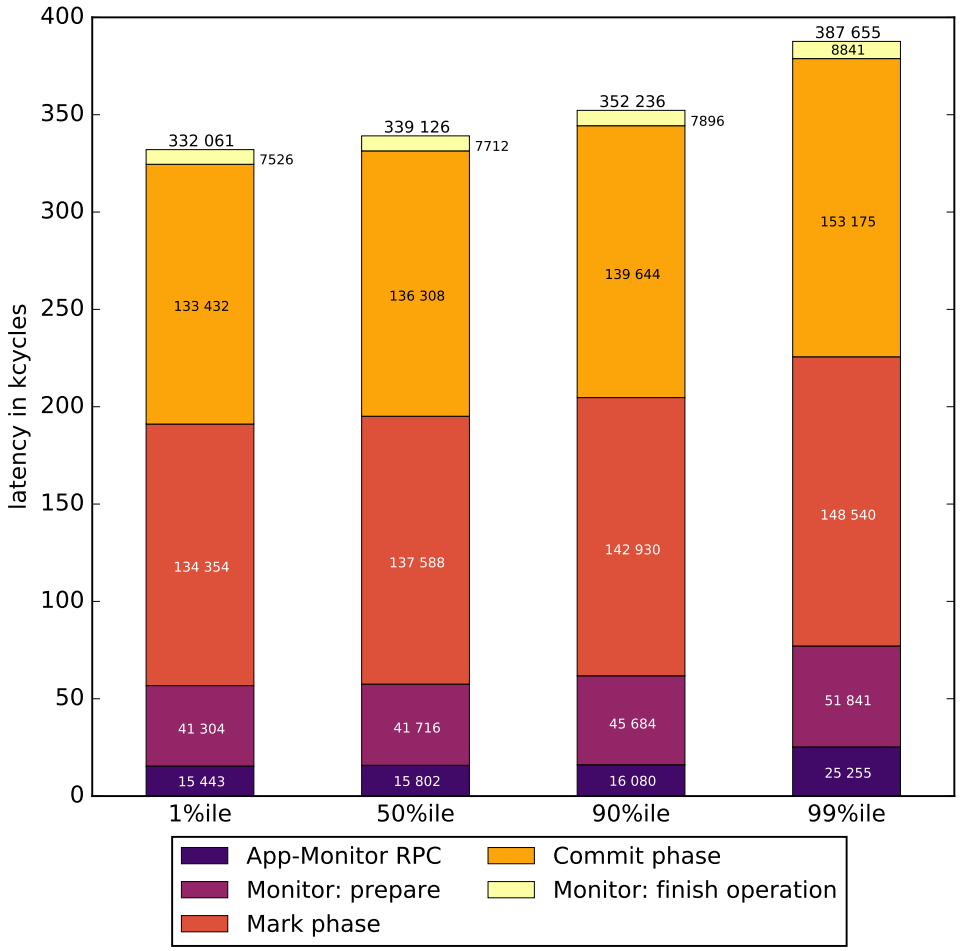


Figure 4.21: *Latency breakdown: revoking a capability with no* foreign *relations*

Revoking a *foreign* copy of a capability The next revoke setting we analyze is calling REVOKE on a foreign copy of a capability. For this experiment, for each measurement, we allocate a RAM capability on one node, the *alloc node*, forward the capability to another node, the *revoke node*, which creates ten copies of the received capability, and then revokes the received capability.

In this situation, before we can revoke the capability, we need to move the capability's ownership to the node on which the revoke is requested. We expect that retrieving ownership is comparable to giving away ownership

in the case where we delete the last local copy of a capability for which foreign copies exist. Looking at the latency breakdown for that variant of delete, cf. figure 4.14, we see that giving away ownership takes roughly 80 kcycles given a mapping database with 4096 synthetic capabilities. Given the experimental design, we expect the two-phase commit to show a similar latency to the case where we revoke a capability with no remote relations, as there will be a single capability that needs to be deleted by the two-phase commit, namely the original copy on the *alloc node*. Putting the pieces together, we predict that the median latency for REVOKE, with a mapping database with 4096 synthetic capabilities, in this situation can be approximated by

$$Latency_{\text{revoke remote copy}} \approx Latency_{\text{move}} + 10 \times Latency_{\text{delete}} + 2 \times Latency_{\text{BC}}$$

Substituting approximate median latencies,

$$\begin{aligned} Latency_{\text{delete}} &\approx 4100\,\text{cycles} \\ Latency_{\text{move}} &\approx 80\,000\,\text{cycles} \\ Latency_{\text{BC}} &\approx 135\,000\,\text{cycles} \end{aligned}$$

we get an estimated latency of

$$\begin{aligned} Latency_{\text{revoke remote copy}} &\approx 80\,\text{kcycles} + 10 \times 4100\,\text{cycles} + 2 \times 135\,\text{kcycles} \\ &= 391\,\text{kcycles} \end{aligned}$$

for revoking a remote copy of a capability.

In figure 4.22, we give the latency for calling REVOKE on a foreign copy of a capability. The figure omits 74 outliers with latencies up to 2.7 Mcycles for some mapping database sizes. Looking at the results for a mapping database which contains 4096 synthetic capabilities, we see that the measured median latency of approximately 538 kcycles is higher than our prediction of 391 kcycles.

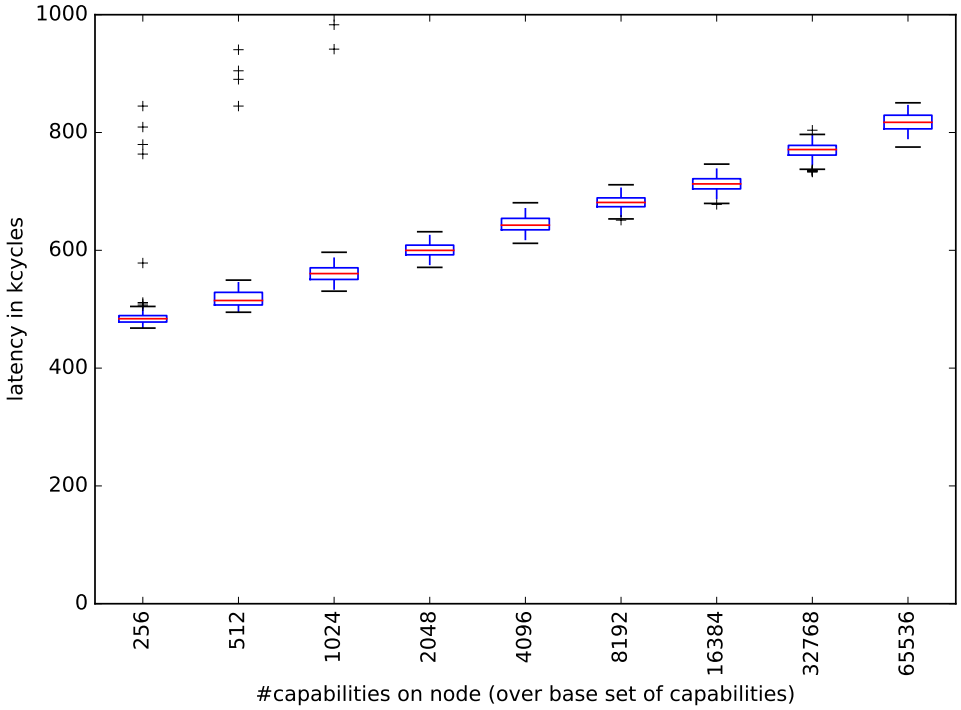


Figure 4.22: *Revoking a* foreign *copy of a capability*

Given the latency breakdown in figure 4.23, we can immediately spot that retrieving a capability's ownership has a median latency of 197 kcycles rather than our prediction of 80 kcycles. Inspecting the ownership move operation, CHOWN, cf. algorithm 4, we see that this operation requires an atomic update of the ownership state for all capability copies in the system. The current implementation in Barrelfish does this atomic update by first doing a unicast to the new owner who then does a broadcast to set the new ownership information on all copies of the capability. Currently, in the case where we give away the ownership when deleting the last local copy of a capability, Barrelfish does not require the ownership update broadcast to complete before completing the DELETE.

For REVOKE however, Barrelfish requires that the ownership update broadcast completes before the revoke can continue. The latency of the "retrieve ownership" phase shown in figure 4.23 therefore is composed of a unicast

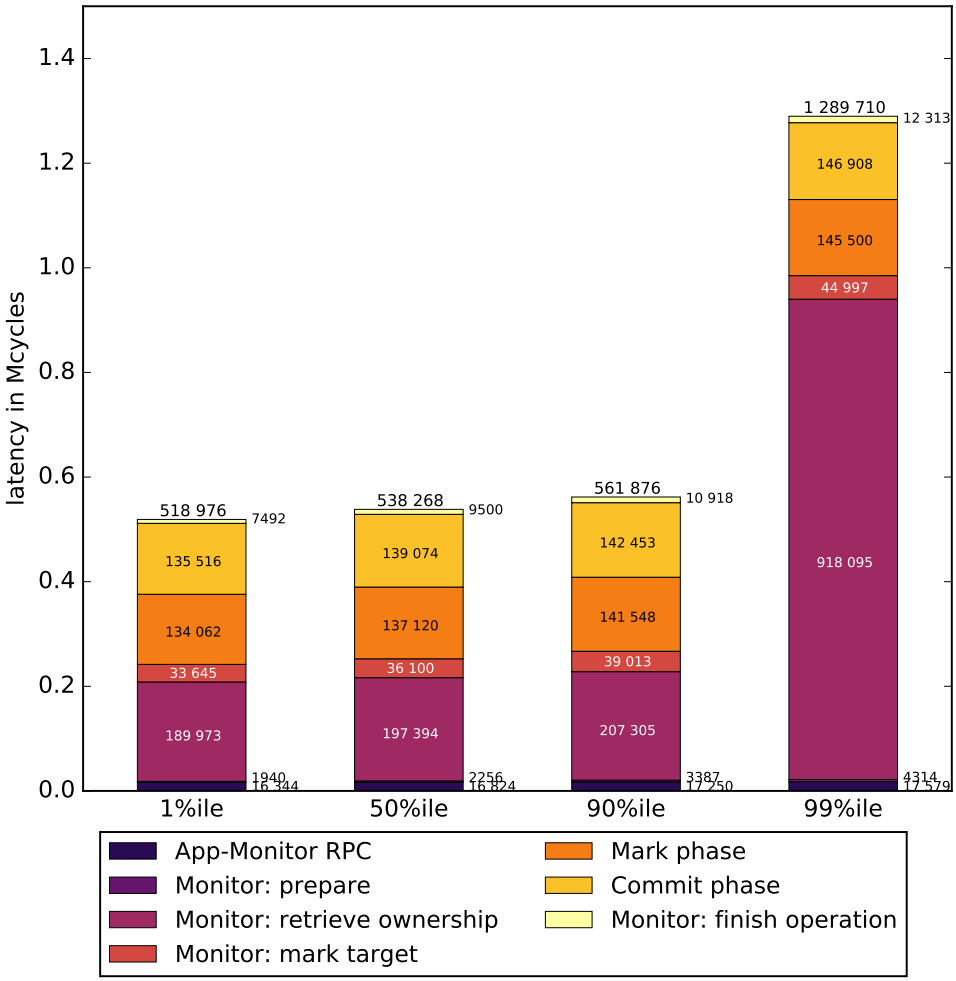


Figure 4.23: *Latency breakdown: revoking a* foreign *copy of a capability*

to the old owner informing it that we acquire the ownership of the given capability, followed by a broadcast from our node. The unicast takes roughly 80 kcycles, as shown by the latency breakdown in figure 4.14, and we have established that the median latency of a broadcast is approximately 135 kcycles.

If we substitute the new expected latency for a full ownership move,

$$\begin{aligned} Latency_{\text{move}} &= Latency_{\text{giveaway}} + Latency_{\text{BC}} \approx 80\,\text{kcycles} + 135\,\text{kcycles} \\ &= 215\,\text{kcycles} \end{aligned}$$

in our formula to predict the latency of revoking a remote copy we get

$$\begin{aligned} Latency_{\text{revoke remote}} &\approx 215\,\text{kcycles} + 10 \times 4100\,\text{cycles} + 2 \times 135\,\text{kcycles} \\ &= 526\,\text{kcycles} \end{aligned}$$

which is fairly close to the measured median latency of 538 kcycles.

Comparing the expected and measured latencies for the different parts we see that our approximation is pretty accurate. As expected, the median latency for a broadcast remains stable at roughly 135 kcycles. We somewhat overestimate the cost of marking the target capability, which is the step in which we delete local copies for which we do not have to make special considerations, as the measured median latency of 36 kcycles is lower than the predicted median latency of 41 kcycles. Similarly, our new estimate for retrieving a capability's ownership is a bit too high at 215 kcycles, compared to the measured median latency of 197 kcycles.

Looking at the breakdown, we can also see that the RPC between the benchmark application and the monitor, and the final cleanup in the monitor remain unchanged from the previous experiment at approximately 15 kcycles and 10 kcycles respectively.

Revoking a *local* copy of a capability with *foreign* relations The last experiment we consider for REVOKE is revoking a capability for which many foreign relations exist. In this experiment, we designate one node to be the *bench node*. For each measurement, the *bench node* first allocates a RAM capability, which is forwarded to a number of other nodes in the system, which we call the *copy nodes*. Each of those nodes creates ten copies of the received capability and signals that it is done. Once all the nodes have informed the *bench node* that they have created copies, the bench node calls revoke on the original copy of the capability and measures the latency. As we run our experiment with only two copy nodes, we expect to see revoke

latencies that look very similar to the first revoke experiment, as the cost of deleting ten copies will be largely masked by the cost of our broadcasts.

A tentative prediction of the latency can be given as

$$Latency_{\text{revoke with remote relations}} \approx 10 \times Latency_{\text{delete}} + 2 \times Latency_{\text{BC}}.$$

Substituting the previously given median latencies for nodes with mapping databases with 4096 synthetic capabilities, we get

$$\begin{aligned} Latency_{\text{revoke with remote relations}} &\approx 10 \times 4100\,\text{cycles} + 2 \times 135\,\text{kcycles} \\ &= 311\,\text{kcycles}. \end{aligned}$$

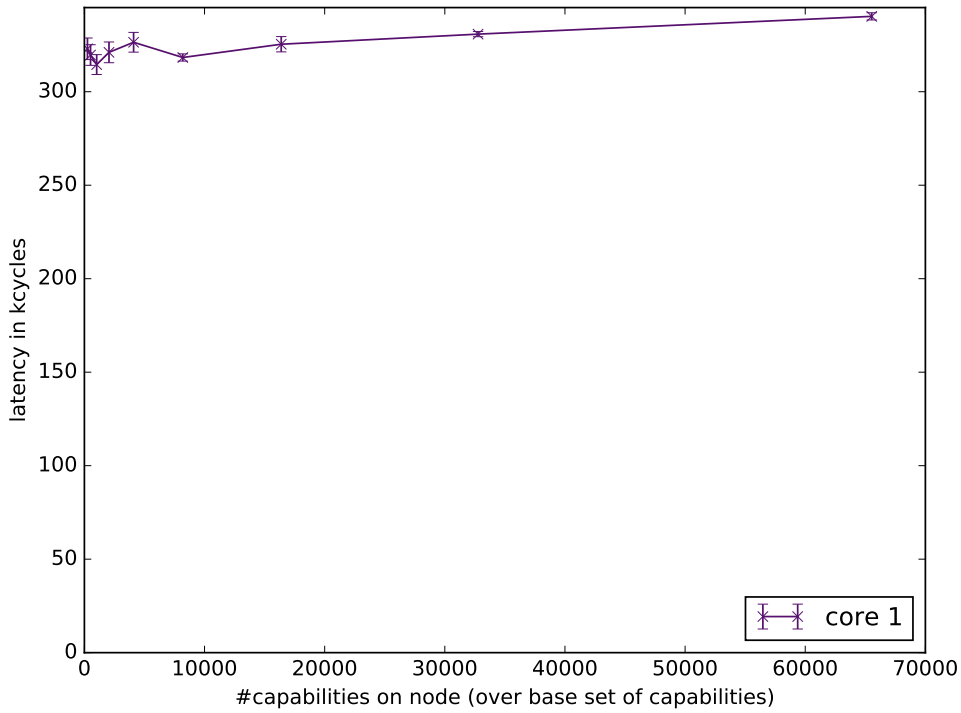


Figure 4.24: *Revoking a* local *copy of a capability with* foreign *relations*

As we can see in figure 4.24, our predicted median latency of 311 kcycles for a mapping database with 4096 synthetic capabilities is very close to the measured latency of revoke which is largely unaffected by mapping database size, and remains relatively stable between 300 kcycles and 350 kcycles.

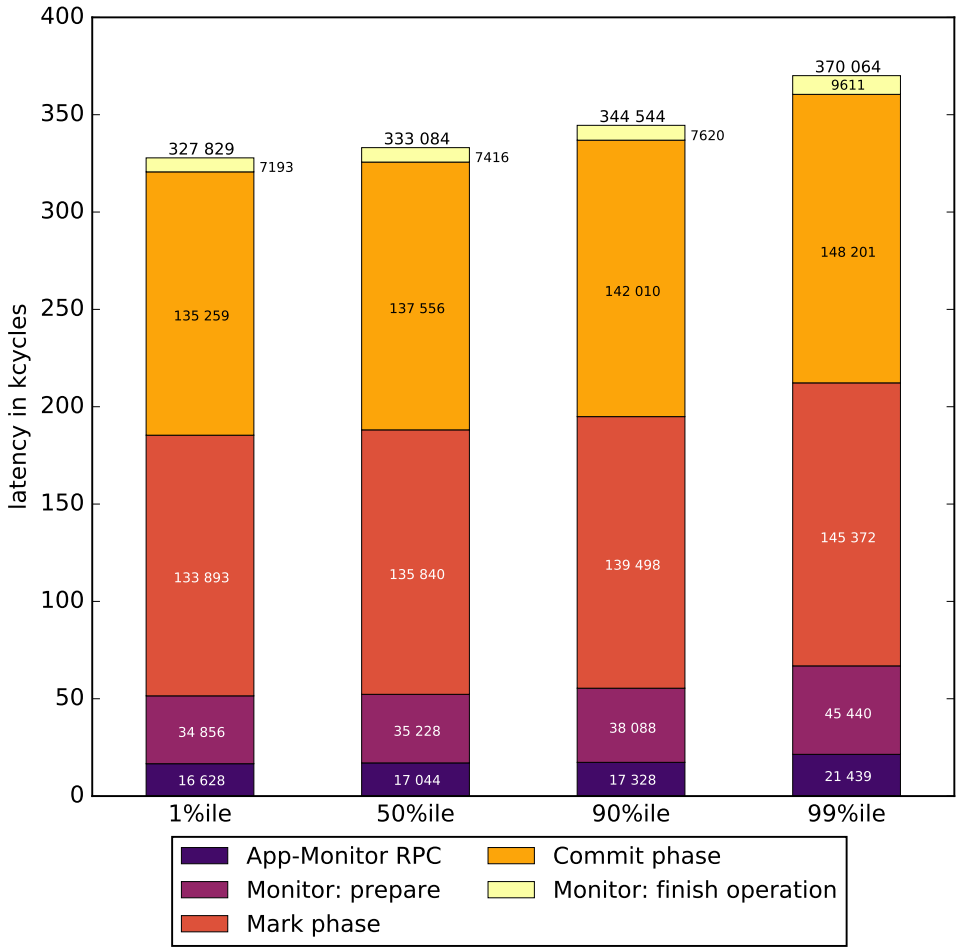


Figure 4.25: *Latency breakdown: revoking a* local *copy of a capability with* foreign *relations*

We also see that the latency breakdown for revoke in the presence of remote relations of the target capability, figure 4.25, looks very similar to the latency breakdown for revoke with a target capability with no remote relations, as shown in figure 4.21.

The latency breakdown also closely matches our prediction, with each broadcast, "Mark phase" and "Commit phase", having a median latency of approximately 135 kcycles, and the "prepare" phase in the monitor taking slightly less than our estimated 41 kcycles.

Retype

For RETYPE we consider the following cases:

1. Retyping a capability with no *foreign* copies and no descendants
2. Retyping a capability with *local* descendants
3. Retyping a capability with *foreign* copies

Retyping a capability with no remote relations and no descendants The first case we consider for RETYPE is again the case where the capability does not have any relations on other nodes. This case should give us some insights into the latency characteristics of the actual retype invocation.

For this experiment, we allocate a single 2 MB RAM capability from the memory server for each mapping database size. We then do a thousand iterations of calling RETYPE on that RAM capability. In each iteration i, we retype a 4 kB Frame capability at offset o,

$$o = (i \times 4\,\text{kB}) \bmod 2\,\text{MB} \tag{4.1}$$

from the base of the 2 MB RAM capability. After measuring RETYPE latency, we delete the capability produced by the retype before starting the next benchmark iteration, to make sure that the retype succeeds for each iteration.

We expect to see fairly low latency for this case, as retyping a capability which has no descendants does not require a range query check.

Figure 4.26 shows that the retype invocation has a latency of 4000 cycles to 5000 cycles given a mapping database which contains between 256 and 65536 synthetic capabilities. In the latency breakdown in figure 4.27, we see that inserting the new capability into the mapping database is the

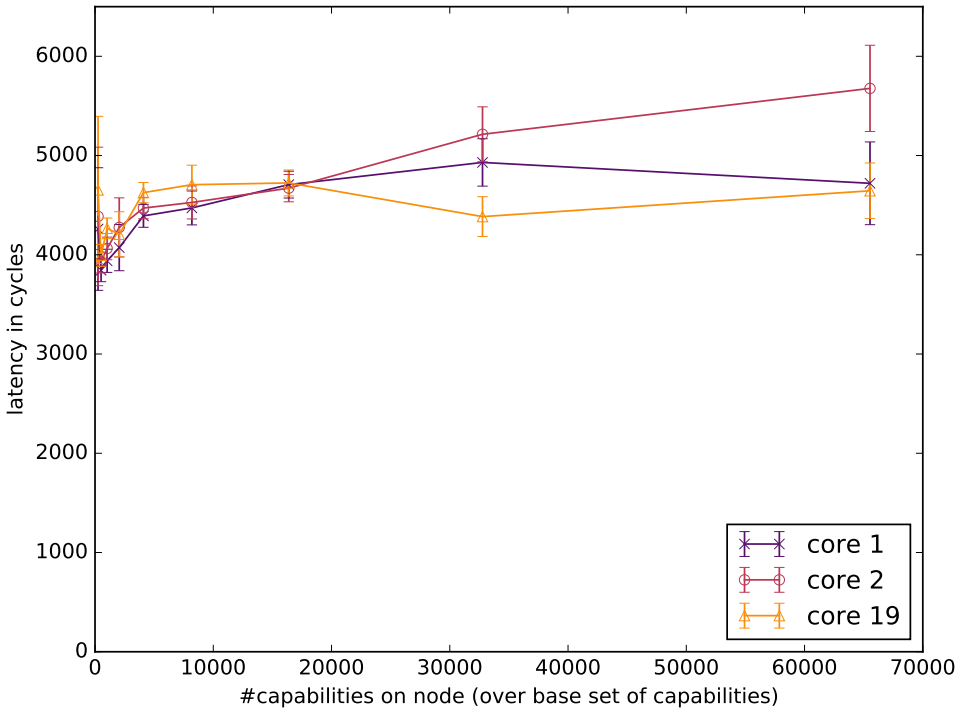


Figure 4.26: *Retype a capability with no* foreign *copies*

operation with the highest latency with a median latency of 1840 cycles for a mapping database with 4096 synthetic capabilities. We also spend a fair amount of effort in zeroing the freshly retyped capabilities. The zeroing has a median latency of 1160 cycles for a mapping database with 4096 synthetic capabilities. The operation with the third-highest latency is checking whether the retype is allowed to proceed, with a median latency of 892 cycles in a mapping database with 4096 synthetic capabilities.

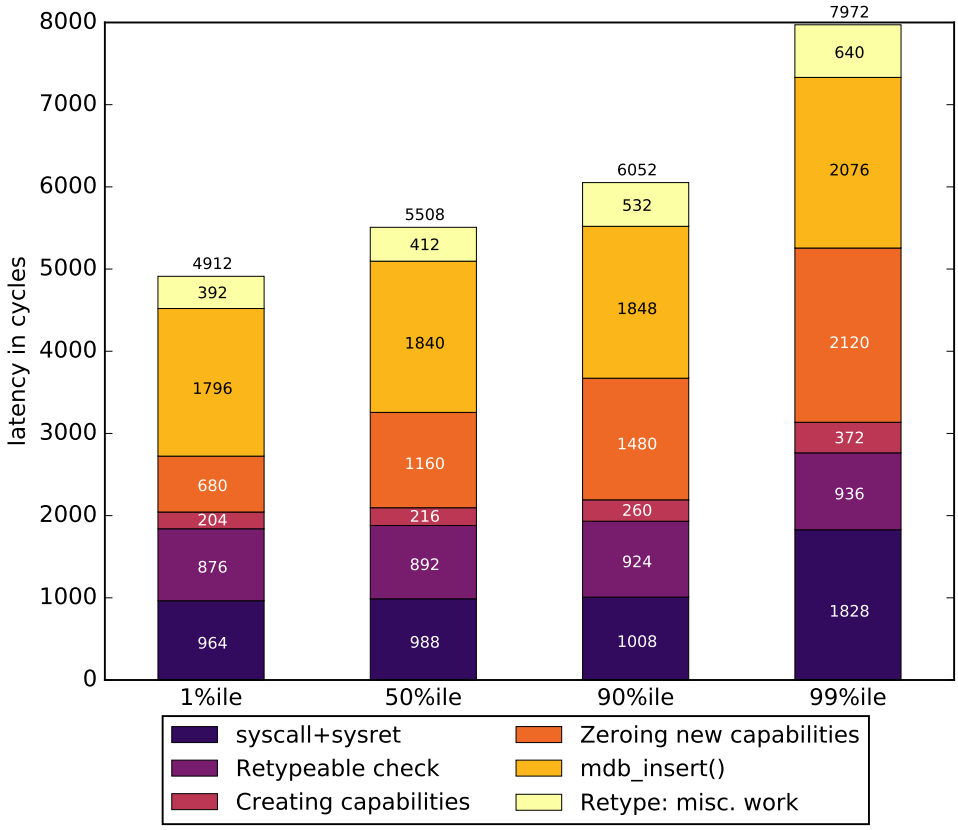


Figure 4.27: *Latency breakdown: retype a capability with no* foreign *copies*

Retyping a capability with local descendants The next case for retype is chosen in such a way that it illustrates the extra cost of allowing subregions of capabilities to be retyped.

In this experiment, we allocate a single 4 MB RAM capability for each mapping database size. Before measuring RETYPE latency in this case, we retype the second half of the 4 MB RAM capability into a 2 MB RAM capability. For each benchmark iterations we use the same strategy as in the previous experiment and retype a 4 kB Frame capability for each iteration while choosing the offset according to the formula given in equation 4.1. Again, each retype result is deleted before starting the next benchmark iteration, so we can retype the same offset multiple times.

Compared to retyping a capability without descendants, the only extra

operation which is required is a range query which is now necessary to make sure it is not possible to create overlapping capabilities by carefully requesting retypes of overlapping subregions of a source capability. Given the performance results for the mapping database operations presented in section 4.5.5, we expect that a range query will have a latency of around 1800 cycles, giving us a predicted latency for this retype of 5800 cycles to 7800 cycles depending on the number of capabilities in the mapping database.

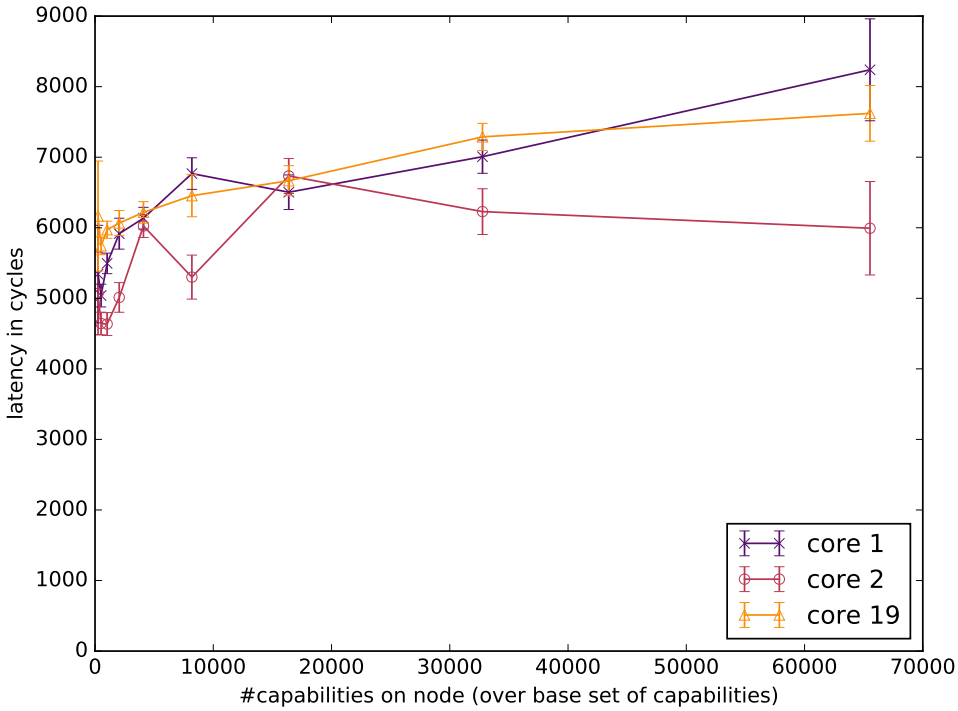


Figure 4.28: *Retype with local descendants*

Figure 4.28 shows the measured latencies for this case, which match our predictions relatively closely, showing a range from approximately 4500 cycles to 8000 cycles. In the latency breakdown shown in figure 4.29, we see that the measured median latency of 1820 cycles for the range query given a mapping database with 4096 synthetic capabilities is very close to our prediction of 1800 cycles. Comparing the latency breakdown for this retype with the latency breakdown for the retype without descendants shown in

figure 4.27, we see that the latencies for the other operations in the CPU driver remain largely unchanged by the presence of local descendants.

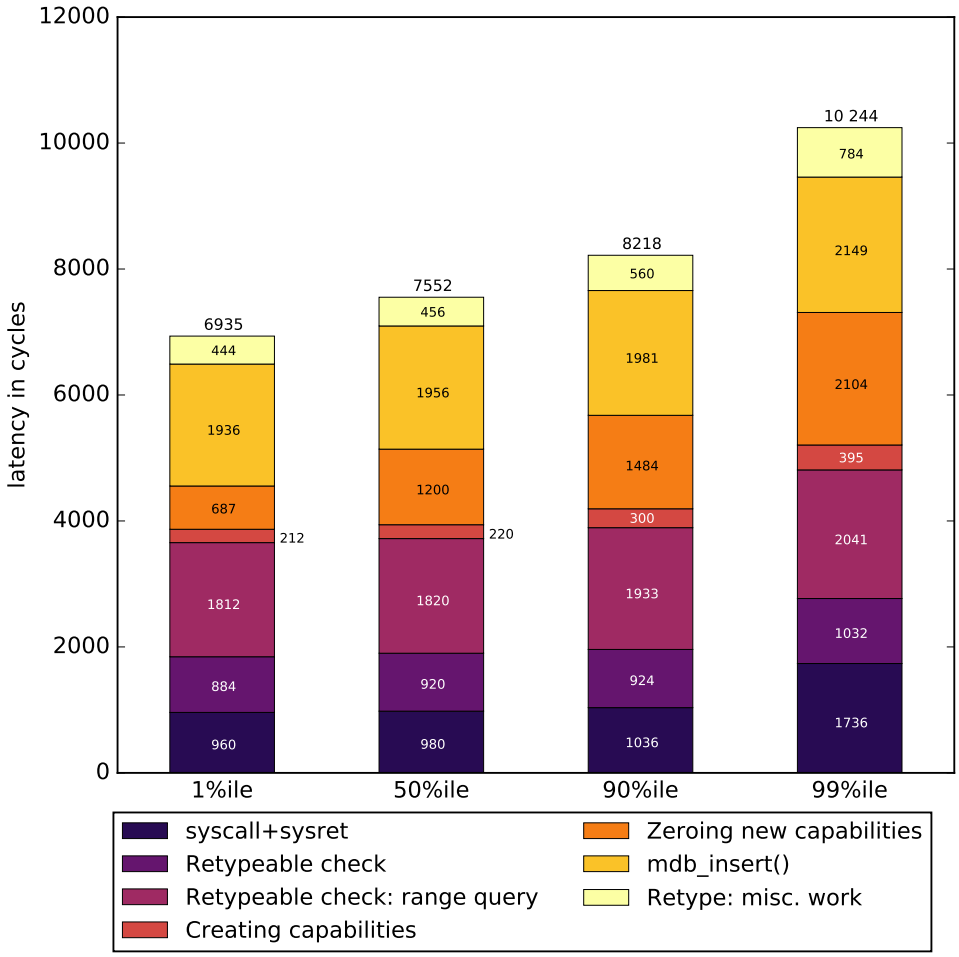


Figure 4.29: *Latency breakdown: retype with local descendants*

Retyping a capability with remote relations The final case of retype is the case where we retype a region of a capability for which remote descendants exist.

The experiment here is setup similarly to the previous retype experiments, with the change that we elect one node to be the node on which we benchmark retype latency which we call the *bench node*. We use the other nodes in the system to create remote descendants of the capability which will be retyped

by the bench node. Again, we allocate a single 4 MB RAM capability for each benchmark round with different mapping database size. After allocating this capability on the bench node, it is forwarded to the other nodes which each create one 4 kB RAM capability at offset $o_r = 2\,\text{MB} + \mathit{nodeid} \times 4\,\text{kB}$ of the 4 MB RAM capability. After these remote descendants are created, the bench node proceeds to do the benchmark iterations with the same strategy as presented for the other retype experiments.

For this experiment, we expect to see a significantly higher retype latency, as we now have to do the retype checks on all nodes of the system which could potentially hold descendants. Because we currently do not keep track of the nodes that actually have remote relations for a given capability, this check is implemented as a broadcast to all the nodes in the system. Given our observations for other operations which require a broadcast, we predict that the retype latency for this experiment as

$$\mathit{Latency}_{\text{retype}} = \mathit{Latency}_{\text{local retype}} + \mathit{Latency}_{BC} + \mathit{Latency}_{\text{monitor RPC}}.$$

Substituting the latency values we observed for the local retype, the monitor RPC, and the broadcast in previous experiments, 7500 cycles, 16 kcycles, and 135 kcycles respectively, we predict that the median latency for retyping a region of a capability with remote relations in a system with a mapping database which contains 4096 synthetic capabilities to be

$$\mathit{Latency}_{\text{retype}} \approx 7500\,\text{cycles} + 16\,\text{kcycles} + 135\,\text{kcycles} = 158.5\,\text{kcycles}$$

We show the median latency of retyping a region of a capability with remote relations on nodes with mapping databases loaded with 512–65536 synthetic capabilities in figure 4.30. We can see that our estimate of 158.5 kcycles is lower than the observed median latency in a system with 4096 synthetic capabilities, which is 213 426 kcycles.

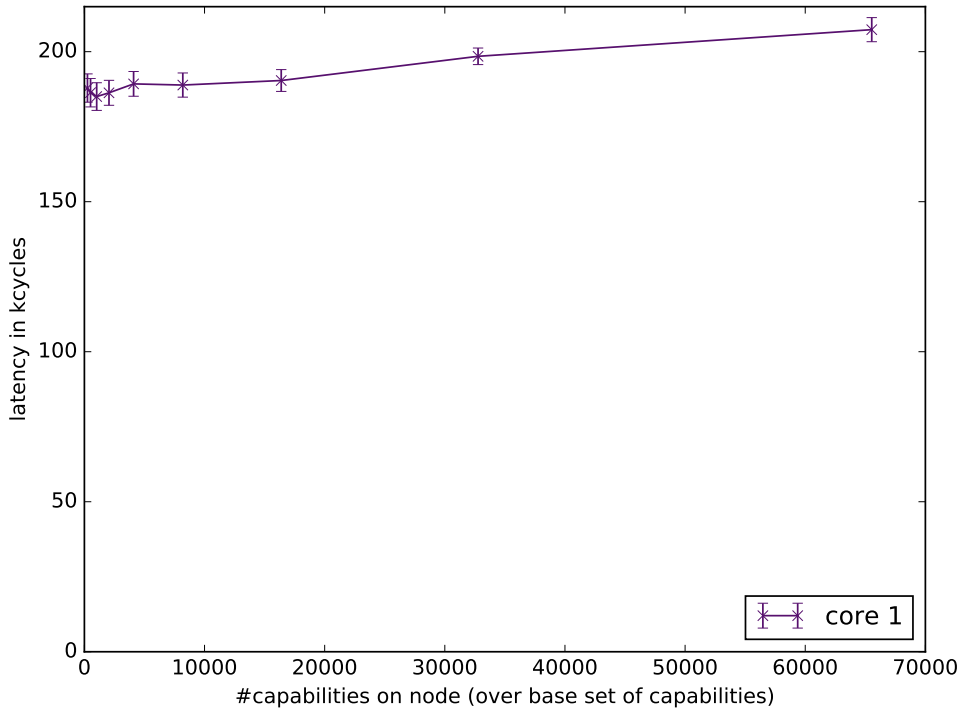


Figure 4.30: *Retype with remote copies*

Looking at the latency breakdown in figure 4.31, we see that rather than our prediction of processing the retype check once locally, we actually do one extra local retype check before doing the broadcast which shows a median latency of 10.5 kcycles which we did not account for in our prediction. An additional 9.5 kcycles, which we did not predict are spent deleting temporary copies of the source and destination root CNodes in the monitor's CSpace. Additionally, we mispredict the cost of the RPC to the monitor by about a factor of two, as the breakdown shows that the median monitor RPC latency is 36 kcycles rather than our prediction of 16 kcycles. Summing up we have identified 40 kcycles which makes up the majority of the difference of 55 kcycles by which our predicted median latency is off of the measured median latency. The remaining 15 kcycles can be attributed to the local retype invocation which the application makes before doing an RPC to the monitor, and various overheads in the monitor.

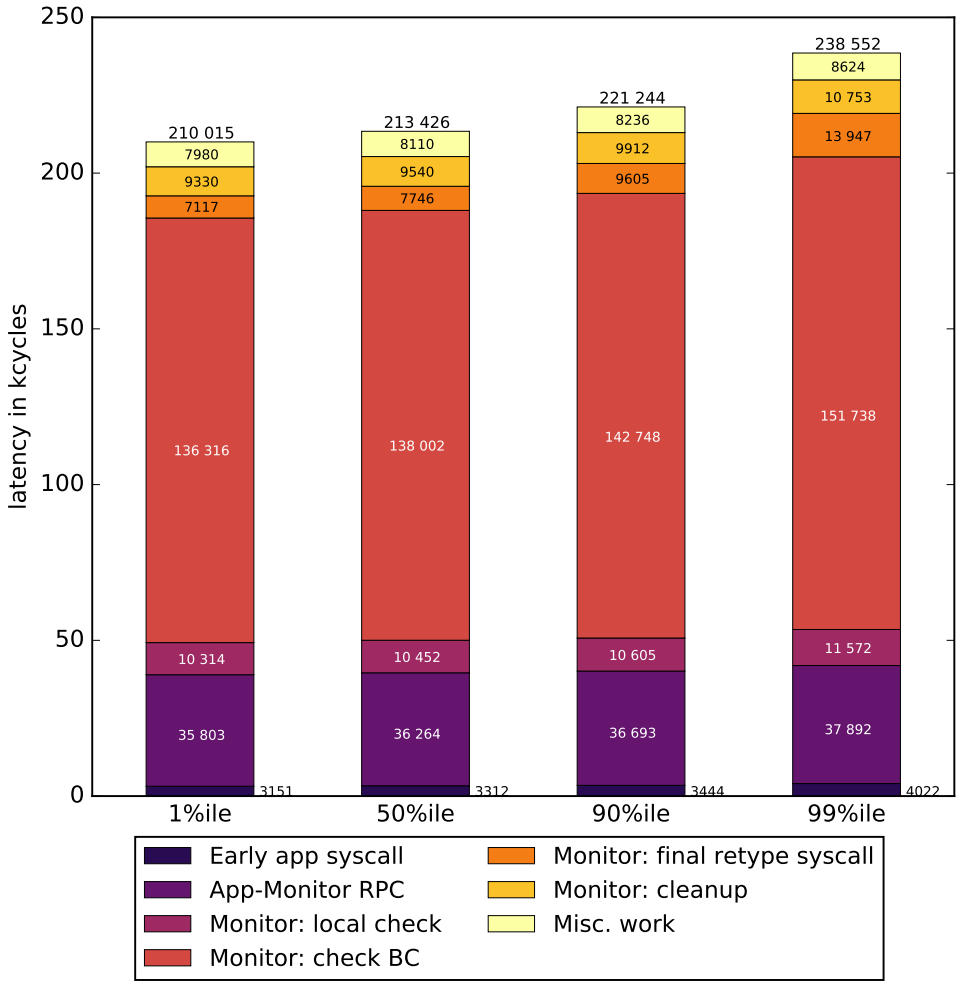


Figure 4.31: *Latency breakdown: retype with remote copies*

After investigating why the monitor RPC latency for retype is a factor of two off from the monitor RPC latency which we observed for revoke, the root cause of this slowdown is the fact that for retype we need to pass two possibly different root CNodes to the monitor, the root CNode for the CSpace which contains the retype source capability, and the root CNode for the CSpace which contains the retype destination slots. Because we need a round trip between the application and the monitor for each capability which we transfer in a RPC, we can explain the increased RPC latency by the need to transfer two capabilities to the monitor instead of one in the revoke case.

We present another latency breakdown for the median retype latency on a node with 4096 synthetic capabilities in the mapping database, where we optimize the case where source capability and destination slots of the retype are located in the same CSpace in figure 4.32. The median latency for the retype with this optimization is 189 kcycles which is 24.5 kcycles lower than the unoptimized case shown in figure 4.31.

We see that the median monitor RPC latency is decreased to 18 kcycles which is much closer to our prediction of 16 kcycles. Additionally, we see that the cost of cleaning up temporary capability copies in the monitor is decreased from 9.5 kcycles to 6 kcycles, and some of the other operations done by the monitor also show slightly lower latencies.

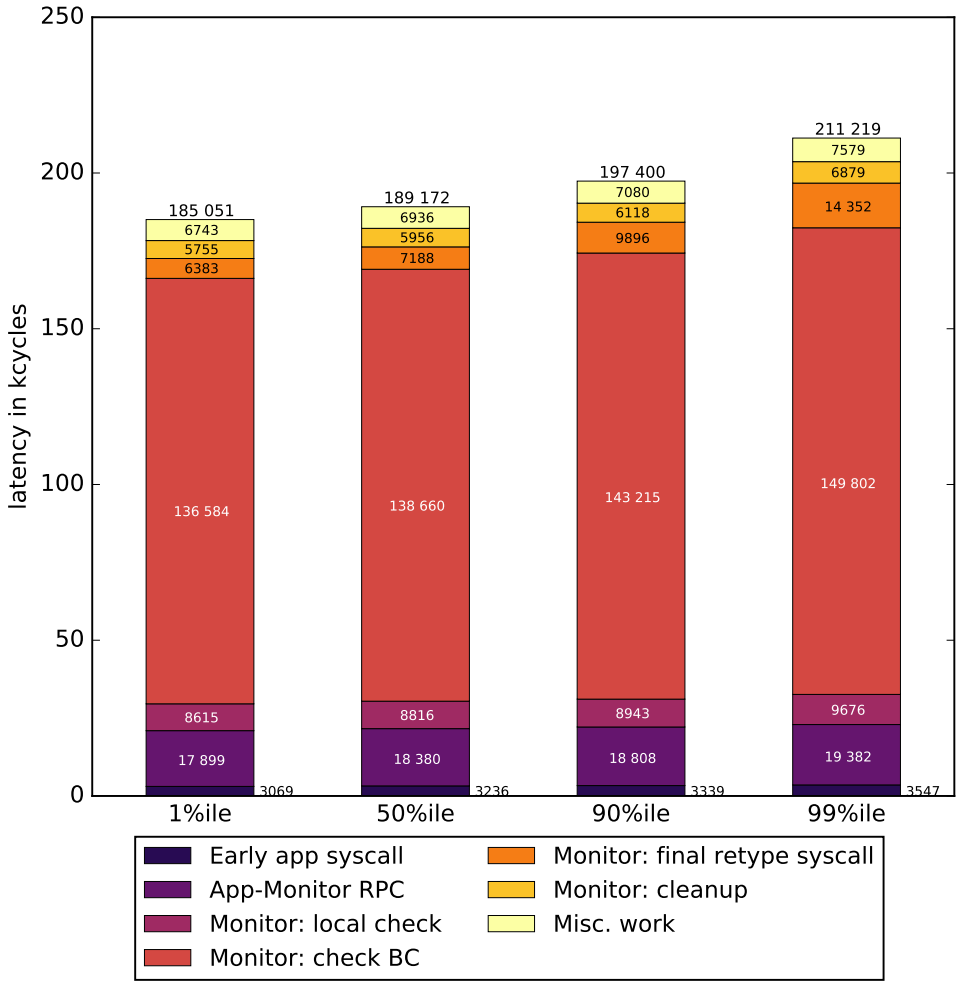


Figure 4.32: *Latency breakdown: retype with remote copies, optimized to not pass two copies of same root CNode to monitor*

5

Formalizing the capablity protocol in TLA+

We now discuss an approach to formalize the algorithms presented in the previous chapter. We present a formal model in TLA+ [Lam02] which aims to capture the high-level behaviour of the system.

The model

We use global state to formalize and verify the assumptions and invariants specified in the previous section. This does not quite correspond to the implementation discussed in section 4.6, but is a good way to check the global invariants of the protocol operations, replicating the operations in a similar fashion to the pseudo code in section 4.2.

MODULE *globaldistops*

EXTENDS *Naturals*

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

First we define constants for the basic components of the capability type system.

All capability types, and the null type specially identified

CONSTANT $CapTypes$

CONSTANT $Null$

Predicates for retype relations and mutability of ownership.

CONSTANTS $RetypeSource(\_)$, $Moveable(\_)$

Null must be a valid type.

ASSUME $Null \in CapTypes$

Use an implicit "NoType" value to indicate a type has no parent.

CONSTANT $NoType$

ASSUME $NoType \notin CapTypes$

With NoType defined, type requirements for RetypeSource become possible:

$FromTypes \triangleq (CapTypes \cup \{NoType\}) \setminus \{Null\}$ Cannot retype from Null

ASSUME $\forall\, t \in CapTypes : RetypeSource(t) \in (FromTypes \setminus \{t\})$

Next, we define requirements for capabilities that refer to addressable resources.

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

CONSTANT Mem

Additional predicates for addressable capabilities

CONSTANTS $Splittable(\_)$, $Addressable(\_)$

Addressable and Moveable must be defined for all types

ASSUME $\forall\, t \in CapTypes : Addressable(t) \in \text{BOOLEAN} \ \wedge Moveable(t) \in \text{BOOLEAN}$

Null is not addressable, while Mem is

ASSUME $Addressable(Null) = \text{FALSE}$

ASSUME $Addressable(Mem) = \text{TRUE}$

Addressability applies to a whole tree of the type forest.

ASSUME $\forall\, t \in CapTypes :$

LET $p \triangleq RetypeSource(t)$

IN $p \neq NoType \Rightarrow (Addressable(t) \equiv Addressable(p))$

Only addressable caps can be split.

ASSUME $\forall\, t \in CapTypes :$

$\land \quad Splittable(t) \in \text{BOOLEAN}$

$\land \quad (\neg Addressable(t)) \Rightarrow (\neg Splittable(t))$

$\land \quad Splittable(t) \Rightarrow Addressable(t)$

The ancestors of a capability are defined as the transitive closure over its retype sources

$Ancestors(t) \triangleq$

IF $RetypeSource(t) = NoType$

THEN $\{\}$

ELSE CHOOSE $s \in \text{SUBSET}\ CapTypes :$

$\land\, RetypeSource(t) \in s$

$\land\, t \notin s$

$\land\, \forall\, parent \in s : \exists\, desc \in (s \cup \{t\}) : RetypeSource(desc) = parent$

The next set of definitions outlines a simple physical address space with contiguous addresses $\in \mathbb{N}_0$ starting at 0. This is a simplification of reality, where addresses do not have to be contiguous, or start at 0, but this makes the model specification and checking a lot simpler and should not change the behavior of the system. The size of the physical address space is a model parameter which can be specified when we check the model. We define a record to represent a region of physical addresses defined by a base and a

size.

CONSTANT $PSpaceSize$

ASSUME $PSpaceSize \in Nat$

$MaxPAddr \triangleq PSpaceSize - 1$

$PSpace \triangleq 0 \,..\, MaxPAddr$

Regions are ranges of PSpace given by a base and size

$Regions \triangleq [base : PSpace,\ size : 1 \,..\, (MaxPAddr + 1)]$

After defining the set of regions, we also define a special identifier for a token region that is not in the set of regions

CONSTANT $NoRegion$

ASSUME $NoRegion \notin Regions$

We define a function to get the set of addresses encompassed by the region

$RegionAddrs(r) \triangleq r.base \,..\, (r.base + r.size - 1)$

Now we define the set of processing cores in the system. The cores have contiguous identifiers $\in \mathbb{N}_0$ starting at 0. Again, these identifiers should not have to be contiguous, or start at 0, but for simplicity of the model we define this to be the case.

CONSTANT $NumCores$

ASSUME $NumCores \in Nat$

$Cores \triangleq 0 \,..\, (NumCores - 1)$

Finally, we define the number of capability slots and concurrent operations in the system as model parameters. We use a variable, *slots*, to represent all capability slots in the system as one big array. Similarly, we use another variable, *operations*, to represent the set of currently active operations in the system.

CONSTANTS $NumSlots$, $NumOps$
VARIABLES $slots$, $operations$

Now we define a record which represents a capability, an expression that declares two capabilities to be copies if their record fields are equal, and a constructor for a capability record that is the result of a retype operation.

$Caps \triangleq [type : CapTypes, region : Regions \cup \{NoRegion\}]$
$NullCap \triangleq [type \mapsto Null, region \mapsto NoRegion]$
ASSUME $NullCap \in Caps$

$IsCapCopy(cap1, cap2) \triangleq cap1.type = cap2.type \land cap1.region = cap2.region$

$Retyped(cap, region, type) \triangleq [type \mapsto type, region \mapsto region]$

Next, we define a formula that returns true iff the capability record *child* is a direct descendant of the capability record *ancestor*.

$$
\begin{aligned}
&IsAncestor(child, ancestor) \triangleq \\
&\quad \lor \land child.type = ancestor.type \\
&\qquad \land Addressable(child.type) \\
&\qquad \land Splittable(child.type) \\
&\qquad \land child.region \neq ancestor.region \\
&\qquad \land RegionAddrs(child.region) \subseteq RegionAddrs(ancestor.region) \\
&\quad \lor \land ancestor.type \in Ancestors(child.type) \\
&\qquad \land \neg Addressable(child.type) \\
&\quad \lor \land ancestor.type \in Ancestors(child.type) \\
&\qquad \land Addressable(child.type) \\
&\qquad \land RegionAddrs(child.region) \subseteq RegionAddrs(ancestor.region)
\end{aligned}
$$

We also define a formula that returns true iff source capability *cap*, target region *region* and target type *type* form a valid retype operation. The last part of the formula checks that there is no capability referring to a region which overlaps the target region except the source capability and its ancestors.

$$
\begin{array}{l}
CanRetype(cap,\ region,\ type) \triangleq \\
\quad \land\ type \in CapTypes \\
\quad \land\ region \neq NoRegion \Rightarrow RegionAddrs(region) \subseteq PSpace \\
\quad \land\ \lor\ RetypeSource(type) = cap.type \\
\qquad \lor\ cap.type = type \land Splittable(type) \\
\quad \land\ IsAncestor(Retyped(cap,\ region,\ type),\ cap) \\
\quad \land\ \forall\, s \in \text{DOMAIN}\ slots : \\
\qquad \text{LET}\ scap \triangleq slots[s].cap \\
\qquad\qquad\ \ rcap \triangleq Retyped(cap,\ region,\ type) \\
\qquad \text{IN}\quad \lor\ scap.type = Null \\
\qquad\qquad\ \ \lor\ IsAncestor(rcap,\ scap) \\
\qquad\qquad\ \ \lor\ (Addressable(rcap.type) \land Addressable(scap.type)) \\
\qquad\qquad\qquad \Rightarrow (RegionAddrs(rcap.region) \cap RegionAddrs(scap.region)) = \{\}
\end{array}
$$

Now, we define a record for the capability slots in the system, as well as a token representing an unowned slot, *NoOwner*, which is not a valid core identifier. For convenience we define slot identifiers $\in \mathbb{N}_0$, starting at 0. Further, we define two record transformations which we will use when specifying the capability operations later.

The record for the capability slots, given in *Slots*, has four elements: (1) the capability, (2) the capability's owner which is either a member of the set of available cores, or the special token *NoOwner* for empty slots, (3) the

location of the slot represented by an element of *Cores*, and (4) the slot's lock.

CONSTANT $NoOwner$
ASSUME $NoOwner \notin Cores$

$SlotIds \triangleq 0 \,..\, (NumSlots - 1)$

$Slots \triangleq [cap : Caps,\ owner : (Cores \cup \{NoOwner\}),$
$\quad location : Cores,\ locked : \text{BOOLEAN}\ ]$

$SlotWithCap(slot,\ cap,\ owner) \triangleq$
$\quad [slot\ \text{EXCEPT}\ !.cap = cap,\ !.owner = owner,\ !.locked = \text{FALSE}]$

$SlotWithNullCap(slot) \triangleq$
$\quad SlotWithCap(slot,\ NullCap,\ NoOwner)$

Further, we specify a number of transformations which affect the whole *slots* variable. There are model steps in which two logically different operations modify the *slots* variable. TLA+ does not allow such model steps, so we introduce the *SetSlotAndUnlockSrc* transformation, which modifies two elements of the slots variable for model steps that need this operation. The *ClearSlot* operation implicitly also unlocks the cleared slot.

$SetSlot(slotid,\ cap,\ owner) \triangleq$
$\quad slots' = [slots\ \text{EXCEPT}\ ![slotid] = SlotWithCap(@,\ cap,\ owner)]$

$SetSlotAndUnlockSrc(slotid,\ cap,\ owner,\ src) \triangleq$
$\quad slots' = [slots\ \text{EXCEPT}$
$\qquad ![slotid] = SlotWithCap(@,\ cap,\ owner),$
$\qquad ![src].locked = \text{FALSE}]$

$CopySlot(destid,\ srcid) \triangleq SetSlot(destid,\ slots[srcid].cap,\ slots[srcid].owner)$

$ClearSlot(slotid) \triangleq SetSlot(slotid,\ NullCap,\ NoOwner)$

$SetSlotState(slotid,\ st) \triangleq slots' = [slots \text{ EXCEPT } ![slotid].locked = st]$

Additionally, we define a predicate that we can use to check whether two slots contain copies of the same capability record.

$IsSlotCopy(sid1,\ sid2) \triangleq$
$\quad IsCapCopy(slots[sid1].cap,\ slots[sid2].cap)$

We now have enough helpers to specify invariants regarding the capability slots in the system. The first invariant regarding capability slots just asserts basic slot properties, such as the size of the slot array, and the type for the values of each slot. Further, it states that only Null capabilities may not have an owner.

$SlotInvariants \triangleq$
$\quad$ Type correctness of slot array
$\quad \land \text{DOMAIN } slots \subseteq SlotIds$
$\quad \land \forall\, sid \in \text{DOMAIN } slots : slots[sid] \in Slots$
$\quad$ Only Null caps may not have an owner
$\quad \land \forall\, sid \in \text{DOMAIN } slots : slots[sid].owner = NoOwner$
$\qquad\qquad\qquad\qquad \equiv slots[sid].cap.type = Null$

The next invariant encompasses the invariants 4.2 and 4.3. This invariant specifies capability ownership in the presence of capability copies. It states that all copies of a capability have to have the same owner, irrespective of slot location and that for each capability a copy exists on the core that holds ownership to that capability.

$$
\begin{array}{l}
\textit{OwnershipInvariants} \triangleq \\
\quad \land \forall\, \textit{sid}1 \in \text{DOMAIN } \textit{slots},\ \textit{sid}2 \in \text{DOMAIN } \textit{slots} : \\
\qquad \textit{IsSlotCopy}(\textit{sid}1,\ \textit{sid}2) \Rightarrow \textit{slots}[\textit{sid}1].\textit{owner} = \textit{slots}[\textit{sid}2].\textit{owner} \\
\quad \land \forall\, \textit{sid} \in \text{DOMAIN } \textit{slots} : \textit{slots}[\textit{sid}].\textit{owner} \neq \textit{NoOwner} \\
\qquad \Rightarrow \exists\, \textit{sid}2 \in \text{DOMAIN } \textit{slots} : \\
\qquad\qquad \textit{IsSlotCopy}(\textit{sid},\ \textit{sid}2) \land \textit{slots}[\textit{sid}2].\textit{owner} = \textit{slots}[\textit{sid}2].\textit{location}
\end{array}
$$

We can also define a temporal invariant which states that the location of each slot is immutable regardless of what operations are executed.

$$
\begin{array}{l}
\textit{SingleLocationProperty} \triangleq \\
\quad \forall\, \textit{sid} \in \text{DOMAIN } \textit{slots} : \textit{slots}'[\textit{sid}].\textit{location} = \textit{slots}[\textit{sid}].\textit{location}
\end{array}
$$

We can express another temporal invariant, stating that a non-null capability cannot be modified without being deleted first.

$$
\begin{array}{l}
\textit{SlotImmutabilityProperty} \triangleq \\
\quad \forall\, \textit{sid} \in \text{DOMAIN } \textit{slots} : \\
\qquad (\textit{slots}[\textit{sid}].\textit{cap}.\textit{type} \neq \textit{Null} \land \textit{slots}'[\textit{sid}].\textit{cap}.\textit{type} \neq \textit{Null}) \\
\qquad\quad \Rightarrow \textit{IsCapCopy}(\textit{slots}'[\textit{sid}].\textit{cap},\ \textit{slots}[\textit{sid}].\textit{cap})
\end{array}
$$

Now we define records and state predicates for the different capability operations. First we define an operation request record for each operation. These requests will be used to define the set of possible operations a user of the system can request.

$$
\begin{array}{ll}
\textit{CopyReq} & \triangleq [\textit{name} : \{\text{“copy”}\},\ \textit{src} : \textit{SlotIds},\ \textit{dest} : \textit{SlotIds}] \\
\textit{RetypeReq} & \triangleq [\textit{name} : \{\text{“retype”}\},\ \textit{src} : \textit{SlotIds},\ \textit{region} : \textit{Regions},
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\quad type \;\; : CapTypes,\ dest : SlotIds] \\
DeleteReq \;\triangleq\; [name : \{\text{“delete”}\},\ target \;\; : SlotIds] \\
RevokeReq \;\triangleq\; [name : \{\text{“revoke”}\},\ target : SlotIds] \\
RequestTypes \;\triangleq\; CopyReq \cup RetypeReq \cup DeleteReq \cup RevokeReq
\end{array}
$$

We then define another record for each operation that is currently running.

$$
\begin{array}{l}
CopyOp \;\triangleq\; [name : \{\text{“copy”}\},\ src : Caps, \\
\qquad\qquad owner : (Cores \cup \{NoOwner\}),\ dest : SlotIds] \\
RetypeOp \;\triangleq\; [name : \{\text{“retype”}\},\ src : Caps,\ region : Regions, \\
\qquad\qquad type \;\; : CapTypes,\ dest : SlotIds] \\
DeleteOp \;\triangleq\; [name : \{\text{“delete”}\},\ target \;\; : SlotIds] \\
RevokeOp \;\triangleq\; [name : \{\text{“revoke”}\},\ target : SlotIds,\ target\_cap : Caps] \\
OperationTypes \;\triangleq\; CopyOp \cup RetypeOp \cup DeleteOp \cup RevokeOp
\end{array}
$$

Next, we define the states an operation can be in, and define the set of new requests as the set of records constructed of all the request types with flag *launched* set to false. Further, we define the set of launched requests as a set of records which have a request record in field *req*, their *launched* flag set to true, and additional record fields *op* and *state* reflecting the operation's type and state respectively. We also define a state predicate on operation requests with a *state* field which is true iff the operation has completed.

$$
\begin{array}{l}
OperationStates \;\triangleq\; \{\text{“running”}, \text{“failed”}, \text{“succeeded”}\} \\
NewRequests \;\triangleq\; [req : RequestTypes,\ launched : \{\text{FALSE}\}] \\
LaunchedRequests \;\triangleq\; [req : RequestTypes,\ launched : \{\text{TRUE}\}, \\
\qquad\qquad op : OperationTypes,\ state : OperationStates] \\
OperationComplete(o) \;\triangleq\; o.state \in \{\text{“failed”}, \text{“succeeded”}\}
\end{array}
$$

We define the set of all operations to be the union of new requests and launched requests. We also define contiguous operation identifiers $\in \mathbb{N}_0$ for simplicity.

$Operations \triangleq NewRequests \cup LaunchedRequests$

$OperationIds \triangleq 0 \,..\, (NumOps - 1)$

With these predicates, we can define an invariant over the *operations* variable stating that the variable reprents an array of operations with array indices $\in$ *OperationIds* and that each array element is an element of the *Operations* set.

$OperationInvariants \triangleq$

$\quad \land$ DOMAIN $operations \subseteq OperationIds$

$\quad \land \forall\, o \in$ DOMAIN $operations : operations[o] \in Operations$

Before we get to the actual operation definitions, we define some helpers.

Get source slot for operation with operation id ‘oid‘

$GetOpSrc(oid) \triangleq$

$\quad$ LET $rq \triangleq operations[o].req$

$\qquad\quad opname \triangleq rq.name$

$\quad$ IN CASE $opname =$ “copy” $\lor\ opname =$ “retype” $\rightarrow rq.src$

$\qquad\quad \square\, opname =$ “delete” $\lor\ opname =$ “revoke” $\rightarrow rq.target$

Lock source slot for operations[oid]

$LockSrcSlot(oid) \triangleq$ LET $src \triangleq GetOpSrc(oid)$

$\qquad\qquad\qquad\quad$ IN $SetSlotState(src,$ TRUE$)$

Unlock source slot for operations[oid]

$$UnlockSrcSlot(oid) \triangleq \text{LET } src \triangleq GetOpSrc(oid) \text{ IN } SetSlotState(src, \text{FALSE})$$

Next we define a predicate that we can use to check whether a particular operation can transition from pending to launched.

$$\begin{array}{l} CanStart(req) \triangleq \\ \quad \text{CASE } req.name = \text{“copy”} \vee req.name = \text{“retype”} \rightarrow \\ \qquad \wedge\ req.src \in \text{DOMAIN } slots \\ \qquad \wedge\ slots[req.src].locked = \text{FALSE} \\ \qquad \wedge\ req.dest \in \text{DOMAIN } slots \\ \quad \square \quad req.name = \text{“delete”} \vee req.name = \text{“revoke”} \rightarrow \\ \qquad req.target \in \text{DOMAIN } slots \wedge slots[req.target].locked = \text{FALSE} \end{array}$$

We also define a state function which we can use to make an operation out of an unlaunched operation request.

$$\begin{array}{l} MkRequestOp(req) \triangleq \\ \quad \text{CASE } req.name = \text{“copy”} \quad \rightarrow [name \mapsto req.name, \\ \qquad\qquad\qquad\qquad\quad src \mapsto slots[req.src].cap, \\ \qquad\qquad\qquad\qquad\quad owner \mapsto slots[req.src].owner, \\ \qquad\qquad\qquad\qquad\quad dest \mapsto req.dest] \\ \quad \square \quad req.name = \text{“retype”} \rightarrow [req \text{ EXCEPT } !.src = slots[@].cap] \\ \quad \square \quad req.name = \text{“delete”} \rightarrow req \\ \quad \square \quad req.name = \text{“revoke”} \rightarrow [name \mapsto req.name, \\ \qquad\qquad\qquad\qquad\quad target \mapsto req.target, \\ \qquad\qquad\qquad\qquad\quad target\_cap \mapsto slots[req.target].cap] \end{array}$$

Additionally we define a state function which we can use to transition one operation slot from not *launched* to *launched*.

$$
\begin{array}{l}
StartOp(oid) \triangleq \\
\quad \land \quad \neg operations[oid].launched \\
\quad \land \quad CanStart(operations[oid].req) \\
\quad \land \quad LockSrcSlot(oid) \\
\quad \land \quad operations' = [operations \text{ EXCEPT } ![oid] = [\\
\qquad\qquad\qquad\qquad req \mapsto @.req, \\
\qquad\qquad\qquad\qquad launched \mapsto \text{TRUE}, \\
\qquad\qquad\qquad\qquad op \mapsto MkRequestOp(@.req), \\
\qquad\qquad\qquad\qquad state \mapsto \text{“running”}]]
\end{array}
$$

Now we define a couple more helper functions which we can use to transition a completed operation to another state, such as “succeeded”.

$$
\begin{array}{l}
SetOpState(o,\ state) \triangleq \\
\quad operations' = [operations \text{ EXCEPT } ![o].state = state] \\
FailOp(o) \triangleq SetOpState(o, \text{“failed”}) \\
SucceedOp(o) \triangleq SetOpState(o, \text{“succeeded”})
\end{array}
$$

With all of these definitions, we can now specify all the distributed capability operations. We first specify the copy operation. Copy is relatively simple. We extract the source capability for the copy and its owner from the operation request that we are executing in this model step. We first check whether the destination slot is occupied or the source slot contains a *Null* capability. If either of these conditions is true we fail the operation. Otherwise we create a copy of the source capability in the destination slot and complete the operation successfully.

```
RunCopy(o) ≜
  LET op    ≜ operations[o].op
      src   ≜ slots[operations[o].req.src].cap
      owner ≜ slots[operations[o].req.src].owner
  IN  CASE slots[op.dest].cap.type ≠ Null ∨ src.type = Null
            Fail copy when src empty or dest occupied
           → ∧ UnlockSrcSlot(o)
             ∧ FailOp(o)
        □  OTHER
           → ∧ SetSlotAndUnlockSrc(op.dest, src, owner, GetOpSrc(o))
             ∧ SucceedOp(o)
```

Next, we specify the retype operation. Retype does a number of checks, the failure of any of these fails the operation. After the checks we create the new capability in the destination slot and complete the operation.

```
RunRetype(o) ≜
  LET op ≜ operations[o].op
           Fail when dest occupied
  IN  CASE ∨ slots[op.dest].cap.type ≠ Null
             Fail if src deleted concurrently
           ∨ slots[operations[o].req.src].cap.type = Null
             Fail if retype request is not valid
           ∨ CanRetype(op.src, op.region, op.type)
           → UnlockSrcSlot(o) ∧ FailOp(o)
        □  OTHER
           → ∧ LET retyped ≜ Retyped(op.src, op.region, op.type)
                 IN  SetSlotAndUnlockSrc(op.dest, retyped,
```

$slots[op.dest].location,$
$GetOpSrc(o))$

$\land\ SucceedOp(o)$

The third operation we specify is delete. Delete requires a few different cases.

First off we handle the "easy" cases, such as deleting a *Null* slot, deleting a non-owned copy, or deleting a capability of which copies exist on the same core. Next we handle the case where we delete a non-moveable copy which has no local copies. In this case, we need to delete all non-owned copies as well as the copy on which delete was called.

Then, we consider the case where we can move ownership of the remaining copies of the deleted capabilities to some other core. In that case, we pick an arbitrary copy of the deleted capability and take that copy's location as the new owner for all remaining copies. Finally, we handle the case where we delete the last existing copy of a capability in the system. This case looks really simple in our model because we do not model the CSpace on each core as CNode capabilities which would require special attention when their last copy is deleted. Also, we cannot express the fact that deleting the last copy of a capability should trigger a memory reclamation process, because we do not take that part of the system into consideration in the model.

$RunDelete(o) \triangleq$
LET $op \triangleq operations[o].op$
$slotid \triangleq op.target$
$slot \triangleq slots[op.target]$
Deleting Null slots is OK and a no-op
IN CASE $\lor\ slot.cap.type = Null$

$$
\begin{array}{ll}
 & \rightarrow \quad UnlockSrcSlot(o) \land SucceedOp(o) \\
 & \quad \text{Non-owned, just delete} \\
\Box & \quad \lor\, slot.location \neq slot.owner \\
 & \quad \text{Have copies on same core, just delete} \\
 & \quad \lor\, (\exists\, s \in \text{DOMAIN } slots : \\
 & \qquad \land\, s \neq slotid \\
 & \qquad \land\, slots[s].location = slot.location \\
 & \qquad \land\, IsSlotCopy(s,\, slotid)) \\
 & \quad \rightarrow ClearSlot(slotid) \land SucceedOp(o) \\
 & \quad \text{Cannot move, delete all copies} \\
\Box & \quad \neg Moveable(slot.cap.type) \\
 & \quad \rightarrow\, \land\, slots' = [s \in \text{DOMAIN } slots \mapsto \text{IF } IsSlotCopy(s,\, slotid) \\
 & \qquad\qquad\qquad \text{THEN } SlotWithNullCap(slots[s]) \\
 & \qquad\qquad\qquad \text{ELSE } slots[s]] \\
 & \qquad \land\, SucceedOp(o) \\
 & \quad \text{Migrate ownership and delete} \\
\Box & \quad (\exists\, s \in \text{DOMAIN } slots : \\
 & \qquad \land\, s \neq slotid \\
 & \qquad \land\, slots[s].location \neq slot.location \\
 & \qquad \land\, IsSlotCopy(s,\, slotid)) \\
 & \quad \rightarrow \exists\, s \in \text{DOMAIN } slots : \\
 & \qquad \land\, s \neq slotid \\
 & \qquad \land\, IsSlotCopy(s,\, slotid) \\
 & \qquad \land\, slots' = [c \in \text{DOMAIN } slots \mapsto \\
 & \qquad\qquad\qquad \text{CASE } c = slotid \rightarrow SlotWithNullCap(slot) \\
 & \qquad\qquad\qquad\quad \Box \quad IsSlotCopy(c,\, slotid) \rightarrow \\
 & \qquad\qquad\qquad\qquad\quad [slots[c] \text{ EXCEPT } !.owner = slots[s].location] \\
 & \qquad\qquad\qquad\quad \Box \quad \text{OTHER } \rightarrow slots[c]] \\
 & \qquad \land\, SucceedOp(o)
\end{array}
$$

```
                Delete last copy of a cap
  □   OTHER  → ClearSlot(slotid) ∧ SucceedOp(o)
```

Finally, we specify revoke. Technically, a revoke is just a delete for each copy and descendant of the revoked capability, but due to constraints in TLA+ – namely, that each variable can only be modified once per model step – we do not model revoke as a series of delete operations. The first case for revoke handles all possible failures for a revoke. Those are either that the we try to revoke a *Null* capability or that we try to revoke a non-moveable capabilty on a non-owned copy. The second case is a failure, because we cannot satisfy the post-condition of revoke – i.e. the only remaining capability that refers to the resource (or parts of it) that exists in the system is the one that we call revoke on – and the system invariants that state that non-moveable capabilities cannot change ownership and that the owning core must always hold at least one copy of an owned capability.

```
RunRevoke(o) ≜
 LET op      ≜ operations[o].op
     slotid  ≜ op.target
     slot    ≜ slots[op.target]
 IN  CASE ∨ slot.cap.type = Null
          ∨ slot.location ≠ slot.owner ∧ ¬Moveable(slot.cap.type)
          → UnlockSrcSlot(o) ∧ FailOp(o)
   □OTHER
     → ∧ slots' = [s ∈ DOMAIN slots ↦
                    CASE s = slotid
                         We handled not moveable in previous case, so always reassign ownership to location of remaining copy here
                          → [slots[s] EXCEPT !.locked = FALSE,
                                             !.owner = slots[s].location]
```

```
                    Delete all copies
            □   IsSlotCopy(s, slotid)
                → SlotWithNullCap(slots[s])
                    Delete all descendants
            □   IsAncestor(slots[s].cap, slot.cap)
                → SlotWithNullCap(slots[s])
            □   OTHER → slots[s]]
    ∧ SucceedOp(o)
```

Now, we have almost everything we need to specify initial and next steps for the model and the theorem we want to check.

Before we come to that, we define a state function which selects an operation that is *launched* and in state "running", and actually executes the state transition function for the operation.

```
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)
```

We define one last state function that we can use to replace a completed operation with a new operation from the pool of available operations

---

$$
\begin{array}{l}
ResetOp(o) \triangleq \\
\quad \land \quad operations[o].launched \\
\quad \land \quad operations[o].state \in \{\text{“failed”}, \text{“succeeded”}\} \\
\quad \land \quad \exists\, newop \in NewRequests : \\
\qquad\quad operations' = [operations \text{ EXCEPT } ![o] = newop] \\
\quad \land \quad \text{UNCHANGED } slots
\end{array}
$$

We now specify the initial state, *Init*, for our model. In the initial state, we define all capability slots to be empty except the slot with identifier 0, which contains a *Mem* capability that covers the full range of the physical address space. Every element of the *operations* variable is selected from the pool of available operations arbitrarily.

In the initial state, we specify the location of each slot to be one of the available cores in a round-robin fashion.

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\, slots = [s \in 0\,..\,(NumSlots - 1) \mapsto \\
\qquad\quad \text{IF } s = 0 \\
\qquad\qquad \text{THEN } [cap \mapsto [type \mapsto Mem, \\
\qquad\qquad\qquad\qquad\quad region \mapsto [base \mapsto 0,\ size \mapsto MaxPAddr + 1]], \\
\qquad\qquad\qquad\quad owner \mapsto 0,\ location \mapsto s\%NumCores, \\
\qquad\qquad\qquad\quad locked \mapsto \text{FALSE}] \\
\qquad\qquad \text{ELSE } [cap \mapsto NullCap,\ owner \mapsto NoOwner, \\
\qquad\qquad\qquad\quad location \mapsto s\%NumCores,\ locked \mapsto \text{FALSE}]] \\
\quad \land\, operations \in [0\,..\,(NumOps - 1) \to NewRequests]
\end{array}
$$

We define our model step *Next* and our specification *Spec*.

$$
\begin{array}{l}
Next \triangleq \land \exists\, o \in \text{DOMAIN } operations : \\
\qquad\qquad \lor\ StartOp(o) \\
\qquad\qquad \lor\ CompleteOp(o) \\
\qquad\qquad \lor\ ResetOp(o) \\
\\
Spec \triangleq Init \land \Box[Next]_{\langle slots,\, operations \rangle}
\end{array}
$$

We define one big invariant that combines all the invariants we have specified previously and one big temporal invariant.

$$
\begin{array}{l}
TypeInvariant \triangleq \\
\quad \land\ SlotInvariants \\
\quad \land\ OwnershipInvariants \\
\quad \land\ OperationInvariants \\
\\
SlotProperty \triangleq \\
\quad \land\ SingleLocationProperty \\
\quad \land\ SlotImmutabilityProperty
\end{array}
$$

Finally, we define the following theorem which should hold if our algorithms are correct.

$$\text{THEOREM } Spec \Rightarrow \Box\, TypeInvariant \land \Box[SlotProperty]_{\langle slots \rangle}$$

Checking the model

The model by itself does not assume anything about the number of nodes, and capabilities, or the size of the physical memory which the capabilities refer to.

However, that variant of the model, due to its generality cannot be model-checked in a reasonable amount of time. That is, the model checker did not terminate after running for more than two weeks for a system with 3 bytes of physical memory, 2 cores and 3 capability slots, while running two concurrent capability operations.

A big part of the intractability is hidden in the way we specify the possible retypes in the system. To recall, we allow the system to issue any retype with a target region that has a base address that is within the existing physical addresses of the model and any size less or equal the size of the physical memory. This leads to a lot of unnecessary work, as the model checker cannot, and, indeed, does not, understand the symmetries here. There is a lot of symmetry hidden in those target regions even for our previously mentioned system with 3 bytes of physical memory. Consider the following pairs of retypes

1. retype 1 has target region $[base \mapsto 0,\ size \mapsto 2]$ and retype 2 has target region $[base \mapsto 2,\ size \mapsto 1]$.

2. retype 1 has target region $[base \mapsto 0,\ size \mapsto 1]$ and retype 2 has target region $[base \mapsto 1,\ size \mapsto 2]$.

These two combinations of retypes are clearly symmetric, but there is no easy way to convey that information to the model checker, as this is outside of what is possible to specify as symmetry sets in TLA+. However, we can modify the definition for *Regions* that was given in the previous section.

$$Regions \triangleq [base : PSpace,\ size : 1\ ..\ (MaxPAddr + 1)]$$

The new definition only contains a subset of target regions, namely the ones shown in figure 5.1.

$$Regions \triangleq \{[base \mapsto 0,\ size \mapsto 1],\ [base \mapsto 0,\ size \mapsto 2],\ [base \mapsto 0,\ size \mapsto 3],\ [base \mapsto 1,\ size \mapsto 2],\ [base \mapsto 1,\ size \mapsto 3]\}$$

With this change we can successfully validate our modified model in less than a day (approximately 19 hours) on a single 2x10 Intel Xeon E5-2670 v2 machine.

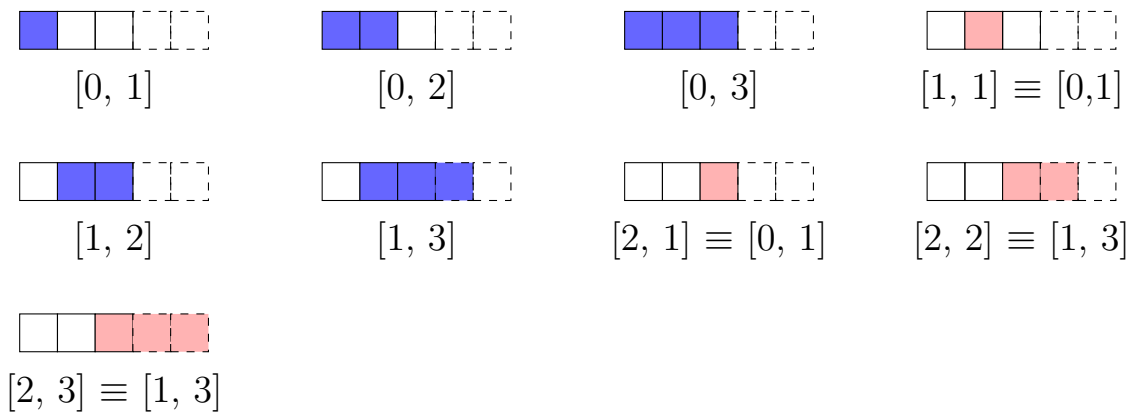


Figure 5.1: *Selected retype target regions in light blue, symmetric ones in light red, with a reference to their partners*

We select the regions shown in figure 5.1 in such a way that we can model check the following retype pairs being executed concurrently:

1. Two retypes with non-overlapping and valid target regions
2. Two retypes with non-overlapping target regions

3. Two retypes with overlapping and valid target regions

4. Two retypes with overlapping target regions

For the case where we have non-overlapping and valid target regions for a pair of retypes, we eliminate all the pairs with two single byte target regions, and the pairing of [0, 2]+[2, 1] which is symmetric to [0, 1]+[1, 2].

For the case of retype pairs with non-overlapping target regions, we reduce the number of "invalid" target regions – i.e. target regions which extend past the limit of our physical address space – to one. This is acceptable because any retype with an invalid target region must fail, disregarding how far past the limit of the address space it reaches. We pick the region [1, 3] as the solitary invalid target region in the reduced set. The combination [0, 1]+[1, 3] stands in for its symmetric – with regard to the retype operation succeeding or failing – counterparts [0, 2]+[2, 2] and [0, 2]+[2, 3]. Additionally there are a number of further combinations with the other single byte target regions, which we do not consider here, as they are directly comparable to either [0, 2]+[2, 2] or [0, 2]+[2, 3].

For retypes with overlapping and valid target regions, we have all the pairs $X+X$ for two retypes with different target slots but the same, valid, target region X, the pairs created by combining two differently-sized target regions starting at byte 0, as well as [0, 2]+[1, 2], and [0, 3]+[1, 2].

For retypes with overlapping, but not necessarily valid target regions we have [0, 2]+[1, 3] and [0, 3]+[1, 3], as well as [1, 3]+[1, 3].

Outlook

Of course the model presented in this chapter does not accurately represent the implementation of the model in Barrelfish.

There are two areas where the model needs to be refined to accurately portray the implementation. First, we need to model the partial mapping database replicas and the messages that are required to execute operations. Second, we need to correctly model the capability spaces as CNode capabilities and implement the model operations to handle dynamic CSpaces.

While it may be possible to express such a more accurate model in TLA+, my impression is that the model checker would struggle with such a specification, and it may be more useful to rewrite the model in PlusCal – or an entirely different logic system.

6

Conclusions

Summary

This dissertation explores a very different style of OS service provision. Demand paging can often have negative impacts on modern applications that rely on fast memory. The virtual address space can be an abstraction which degrades application performance. In Barrelfish, in contrast, an application *knows* when it has insufficient physical memory and must explicitly deal with it. Given current trends in both applications and hardware, we believe that our radically different memory system is worthy of further attention.

In the evaluation of our memory system, we confirm our thesis and show that by turning the classical virtual memory system inside out, we give applications unprecedented freedom in the construction of their virtual address spaces without negatively impacting their performance and, in fact, allowing our system to outperform Linux in some cases..

We then demonstrate that the capability system, which is the foundation of our memory system, can be made scalable with a comparatively simple

protocol which exploits the properties of the message channels available in Barrelfish. We demonstrate that the latency of operations of that capability protocol are acceptable, and present a simple formal model in TLA+ which demonstrates that our protocol does not violate the safety guarantees given by either the memory or capability system.

Directions for Future Work

Barrelfish's memory system has many areas with opportunities for future research. We give a brief outline of a few of those opportunities in this section.

Multiple physical address spaces

We noted in the beginning that our design makes the assumption that there is only a single shared physical address space in a machine. However, with RDMA technologies, such as InfiniBand, we can execute direct reads and writes in another machine's physical address space. Additionally, a lot of large-scale server software runs on a whole rack of machines today, necessitating some form of rack-scale management. We give a more in-depth motivation for the necessity of explicitly managing multiple physical address spaces in a operating system in our 2015 HotOS submission [GZA+15].

Our memory management design should lend itself to such a rack-scale environment where machines can access memory on other machines through one-sided remote reads and writes. The fundamental change which is necessary to make the capability system understand multiple physical address spaces is that each capability has a field which holds an identifier for the physical address space to which the region it refers to belongs. Those

address space identifiers then need to be checked when necessary, e.g. when installing new mappings in a virtual address space. In addition to simply checking the identifiers, the memory system also needs to program the RDMA hardware to allow remote accesses assuming the checks succeed.

Additionally, the distributed capability protocol needs to be extended to work across machine boundaries, where the assumptions we make about message channels, namely that they guarantee in-order, exactly once, FIFO delivery of messages, do not necessarily hold anymore.

One approach that one might take when extending the capability protocol is to create a two-level protocol, where we have a protocol implementation which is more resilient to reordered or lost messages between machines, while we run the protocol presented in chapter 4 inside each machine.

A better capability description language

We have briefly discussed *Hamlet*, Barrelfish's domain specific language for defining capability types in this thesis. While Hamlet currently makes adding new capability types easy, there is a lot of boiler-plate code which needs to be written manually for each capability type. The boiler-plate includes things like small predicate functions which express higher-level properties of capability types, such as whether a capability type is a mapping type or a page table type, as well as a lot of argument marshalling and unmarshalling when implementing invocations for that new capability type.

A research direction which has been discussed, but postponed in favour of more directly OS-related research, as it touches more on programming language design, is to make Hamlet a more full-featured programming language that can express invocations for each capability type and higher-level predicates, and use that improved language to eliminate a lot of the manually written boiler-plate code for capability invocations and predicates.

Hardware acceleration for kernel-based capabilities

An idea that has been discussed a number of times is to see if we can dramatically improve the latency of capability operations by designing custom hardware – most likely on a FPGA which is closely coupled to the CPU – which we can use as a capability offload engine or co-processor.

We never pursued this direction of research until now because available systems which pair CPU and FPGA suffer from latency issues when transferring control from the CPU to the FPGA and back. However, Enzian, `http://enzian.systems`, a research computer platform designed in the Systems Group, will be a system where the CPU and the FPGA are directly connected using the CPU's cache coherency interface, giving applications a low-latency and high-bandwidth link between the CPU and the FPGA. The latency of this link seems to be low enough for a capability co-processor on the FPGA to potentially improve overall capability operation latencies, cf. the results for software-based capability operation latencies in section 4.7.

Additionally, a platform like Enzian opens avenues of making capabilities a mechanism for authorization across multiple physical systems as we could extend the protocol presented in chapter 4 to a cluster of Enzian machines using the FPGA in each node of such a cluster to ensure that over-the-wire capability traffic between nodes is encrypted without imposing high latency overheads from having to do encryption and network traffic processing on the CPU.

Multi-threaded shared-memory applications

One interesting direction for future research is to explore support for multi-threaded shared-memory based applications on a multikernel OS such as Barrelfish.

Barrelfish currently only supports domain spanning on `x86_64`, which is a major limitation of the implementation. The current implementation of domain spanning, as presented in Razvan Damachi's master's thesis [Dam17], does not offer great support for shared address space management. On `x86_64`, we currently statically partition the domain's virtual address space into 512GB chunks, and each core to which the domain spans is assigned one chunk to back local virtual region allocations.

There are a number of options on how to make Barrelfish's library OS aware of the interactions between cores when it comes to spanned domains.

The first variant is to fully embrace the multikernel philosophy and share nothing. This would mean that any page tables that make up the domain's virtual address space need to be fully replicated for each core on which the domain has a dispatcher. The immediate downside of this is that every map request needs to be broadcast to all cores on which the domain has a dispatcher. The domain then has to decide if its virtual address space is identical on all cores. Further the library OS needs to synchronize the shadow page tables etc. on each virtual address space modification.

The second variant is to share everything. In this implementation, the domain's dispatchers would share a single set of page table frames, and a single set of shadow page tables and other user-space data structures. This approach requires carefully implemented thread safe data structures for the shadow page tables, and other user space data structures. In addition to that, we would heavily utilize invocations on foreign capabilities when creating mappings on cores that do not own the particular page table capability involved in the mapping request. As we envision those remote invocations to be proxied to the owning core, this could lead to severe performance degradation, especially in light of having to simultaneously hold locks on the relevant shadow page table entries.

The third option is to share the page table frames, but not the shadow page tables. This is closest to the current system, where we share the page table frames, but do not keep the shadow page tables synchronized between cores. This would need some synchronization to keep track of the actual state of the page table frames in the shadow page tables of all dispatchers, but would be less contention-heavy than the second option, as we only need to synchronize operations that allocate or free virtual address space. Once a region of address space is allocated, the core which requested the allocation gains authority over that region, and can send shadow page table updates to the other cores voluntarily, or reply to an update request.

List of Tables

List of Figures

Bibliography

[ABG+86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation for UNIX Development." *Tech. rep.*, Computer Science Department, Carnegie Mellon University, 1986.

[AJH12] J. Ahn, S. Jin, and J. Huh. "Revisiting Hardware-assisted Page Walks for Virtualized Systems." In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pp. 476–487. IEEE Computer Society, Washington, DC, USA, 2012.

[AL91] A. W. Appel and K. Li. "Virtual Memory Primitives for User Programs." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pp. 96–107. ACM, New York, NY, USA, 1991.

[And93] A. Andersson. "Balanced Search Trees Made Simple." In *Proceedings of the Third Workshop on Algorithms and Data Structures*, WADS '93, pp. 60–71. Springer-Verlag, Berlin, Heidelberg, 1993.

[ARM] ARM Ltd. *Cortex-A9 Technical Reference Manual*. Revision r4p1.

[ARM14] ARM Ltd. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*, 2014. ARM DDI 0406C.c.

[ARM15] ARM Ltd. "ARMv8-A Architecture." Online, 2015. `http://www.arm.com/products/processors/armv8-architecture.php`.

[ARS89] E. Abrossimov, M. Rozier, and M. Shapiro. "Generic Virtual Memory Management for Operating System Kernels." In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pp. 123–136. ACM, 1989.

[Azi14] K. Aziz. "Improving the Performance of Transparent Huge Pages in Linux." `https://blogs.oracle.com/linuxkernel/entry/performance_impact_of_transparent_huge`, 2014.

[BALL90] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. "Lightweight Remote Procedure Call." *ACM Trans. Comput. Syst.*, vol. 8, no. 1, 37–55, 1990.

[BBD+09] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. "The Multikernel: A New OS Architecture for Scalable Multicore Systems." In *Proceedings of the ACM SIGOPS Twenty-Second Symposium on Operating Systems Principles*, SOSP '09, pp. 29–44. Big Sky, Montana, USA, 2009.

[BBM+12] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. "Dune: safe user-level access to privileged CPU features." In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI)*. Hollywood, CA, USA, 2012.

[BCR10] T. W. Barr, A. L. Cox, and S. Rixner. "Translation Caching: Skip, Don't Walk (the Page Table)." In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pp. 48–59. ACM, New York, NY, USA, 2010.

[BDS91] H.-J. Boehm, A. J. Demers, and S. Shenker. "Mostly Parallel Garbage Collection." In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pp. 157–164. 1991.

[BGC+13] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. "Efficient Virtual Memory for Big Memory Servers." In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pp. 237–248. ACM, New York, NY, USA, 2013.

[Bha13] A. Bhattacharjee. "Large-reach Memory Management Unit Caches." In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 383–394. ACM, New York, NY, USA, 2013.

[Boea] H.-J. Boehm. "Conservative GC Algorithmic Overview." `http://www.hboehm.info/gc/gcdescr.html`.

[Boeb] H.-J. Boehm. "GCBench." `http://hboehm.info/gc/gc_bench/`.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. "Extensibility Safety and Performance in the SPIN Operating System." In *Proceedings of the Fifteenth ACM Symposium on Operating*

Systems Principles, SOSP '95, pp. 267–283. ACM, New York, NY, USA, 1995.

[BSSM08] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. "Accelerating Two-dimensional Page Walks for Virtualized Systems." In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pp. 26–35. 2008.

[BWCC+08] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. "Corey: An Operating System for Many Cores." In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pp. 43–57. USENIX Association, Berkeley, CA, USA, 2008.

[Cas13] M. Casey. "Performance Issues with Transparent Huge Pages (THP)." `https://blogs.oracle.com/linux/entry/performance_issues_with_transparent_huge`, 2013.

[CD94] D. R. Cheriton and K. J. Duda. "A Caching Model of Operating System Kernel Functionality." In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94. USENIX Association, Monterey, California, 1994.

[CJ75] E. Cohen and D. Jefferson. "Protection in the Hydra Operating System." In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pp. 141–160. ACM, Austin, Texas, USA, 1975.

[CKD94a] N. P. Carter, S. W. Keckler, and W. 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*, pp. 319–327. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society, San Jose, California, 1994.

[CKD94b] N. P. Carter, S. W. Keckler, and W. J. Dally. "Hardware Support for Fast Capability-based Addressing." In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pp. 319–327. ACM, San Jose, California, USA, 1994.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2 ed., 2001.

[Cor12a] J. Corbet. "AutoNUMA: the other approach to NUMA scheduling." `http://lwn.net/Articles/488709/`, 2012.

[Cor12b] J. Corbet. "NUMA in a hurry." `http://lwn.net/Articles/524977/`, 2012.

[Cor12c] J. Corbet. "Toward better NUMA scheduling." `http://lwn.net/Articles/486858/`, 2012.

[Cor13a] J. Corbet. "NUMA scheduling progress." `http://lwn.net/Articles/568870/`, 2013.

[Cor13b] J. Corbet. "User-space page fault handling." `http://lwn.net/Articles/550555/`, 2013.

[Cor14a] J. Corbet. "2014 LSFMM Summit: Huge page issues." `http://lwn.net/Articles/592011/`, 2014.

[Cor14b] J. Corbet. "NUMA placement problems." http://lwn.net/Articles/591995/, 2014.

[Cor14c] J. Corbet. "Page faults in user space: MADV_USERFAULT, remap_anon_range(), and userfaultfd()." http://lwn.net/Articles/615086/, 2014.

[Cor14d] J. Corbet. "Transparent huge pages in 2.6.38." http://lwn.net/Articles/423584/, 2014.

[CPK08] M. D. Castro, R. D. Pose, and C. Kopp. "Password-Capabilities and the Walnut Kernel." *The Computer Journal*, vol. 51, no. 5, 595–607, 2008.

[Dam17] R.-G. Damachi. *Process Management in a Capability-Based Operating System.* ETH Zurich, 2017. Master's Thesis, http://www.barrelfish.org/publications/ma-damachir-procmgmt.pdf.

[DBMZ08] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. "Hardbound: architectural support for spatial safety of the C programming language." *SIGARCH Comput. Archit. News*, vol. 36, no. 1, 103–114, 2008.

[DBR09] P.-E. Dagand, A. Baumann, and T. Roscoe. "Filet-o-Fish: practical and dependable domain-specific languages for OS development." In *Proceedings of the 5th Workshop on Programming Languages and Operating Systems (PLOS)*. 2009.

[DEE06] P. Derrin, D. Elkaduwe, and K. Elphinstone. *seL4 Reference Manual.* NICTA, 2006. http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf.

[DFF+13] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems." In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pp. 381–394. ACM, Houston, Texas, USA, 2013.

[Dil13] M. Dillon. "Design elements of the FreeBSD VM system - Page Coloring." Online, `https://www.freebsd.org/doc/en/articles/vm-design/page-coloring-optimizations.html`, 2013. Accessed 2015-08-26.

[DY16] G. J. Duck and R. H. C. Yap. "Heap Bounds Protection with Low Fat Pointers." In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pp. 132–142. ACM, Barcelona, Spain, 2016.

[EDE08] D. Elkaduwe, P. Derrin, and K. Elphinstone. "Kernel Design for Isolation and Assurance of Physical Memory." In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, pp. 35–40. ACM, New York, NY, USA, 2008.

[EGK95] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. "AVM: Application-level Virtual Memory." In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, HOTOS '95, pp. 72–. IEEE Computer Society, 1995.

[EKO95] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. "Exokernel: An Operating System Architecture for Application-level Resource Management." In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 251–266. 1995.

[ESG+94] Y. Endo, M. Seltzer, J. Gwertzman, C. Small, K. A. Smith, and D. Tang. "VINO: The 1994 Fall Harvest." *Technical Report TR-34-94*, Center for Research in Computing Technology, Harvard University, 1994.

[Eva15] J. Evans. "Issue #243: Improve interaction with transparent huge pages." `https://github.com/jemalloc/jemalloc/issues/243`, 2015.

[FFB+88] A. Forin, R. Forin, J. Barrera, M. Young, and R. Rashid. "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach." In *In 1988 Winter USENIX Conference*. 1988.

[GARH14] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. "Deployment of Query Plans on Multicores." *Proc. VLDB Endow.*, vol. 8, no. 3, 233–244, 2014.

[Ger12] S. Gerber. "Virtual Memory in a Multikernel." Master's thesis, 2012. Master's Thesis, `http://www.barrelfish.org/publications/gerber-master-vm.pdf`.

[GH12] M. Gorman and P. Healy. "Performance Characteristics of Explicit Superpage Support." In *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10, pp. 293–310. Springer-Verlag, Berlin, Heidelberg, 2012.

[GLD+14] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma. "Large Pages May Be Harmful on NUMA Systems." In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pp. 231–242. USENIX Association, Philadelphia, PA, 2014.

[Gor10a] M. Gorman. "Huge pages." `http://lwn.net/Articles/374424/`, 2010.

[Gor10b] M. Gorman. "Huge pages part 2: Interfaces." `https://lwn.net/Articles/375096/`, 2010.

[GZA+15] S. Gerber, G. Zellweger, R. Achermann, K. Kourtis, T. Roscoe, and D. Milojicic. "Not Your Parents' Physical Address Space." In *15th Workshop on Hot Topics in Operating Systems*, HotOS XV. Kartause Ittingen, Switzerland, 2015.

[Han] D. Hansen. "TLB flushing on x86." `https://www.kernel.org/doc/Documentation/x86/tlb.txt`.

[Han99] S. M. Hand. "Self-paging in the Nemesis Operating System." In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pp. 73–86. USENIX Association, New Orleans, Louisiana, USA, 1999.

[HB] B. Haible and P. Bonzini. "GNU libsigsegv - Handling page faults in user mode." `http://libsigsegv.sourceforge.net/`.

[HC92] K. Harty and D. R. Cheriton. "Application-controlled Physical Memory Using External Page-cache Management." In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pp. 187–197. ACM, New York, NY, USA, 1992.

[HCSO12] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. "Green-Marl: A DSL for Easy and Efficient Graph Analysis." In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pp. 349–362. ACM, New York, NY, USA, 2012.

[HHL+97] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. "The Performance of μ-Kernel-based Systems." In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pp. 66–77. ACM, New York, NY, USA, 1997.

[HP 15] HP Labs. "The Machine." `http://www.hpl.hp.com/research/systems-research/themachine/`, 2015.

[HSH81] M. E. Houdek, F. G. Soltis, and R. L. Hoffman. "IBM System/38 Support for Capability-based Addressing." In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pp. 341–348. IEEE Computer Society Press, Minneapolis, Minnesota, USA, 1981.

[Iii91] J. B. Iii. "A Fast Mach Network IPC Implementation." In *USENIX MACH Symposium*, pp. 1–11. USENIX, 1991.

[Int] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Order Number: 325384-052US.

[Int13] Intel Plc. "Introduction to Intel® Memory Protection Extensions." `http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`, 2013.

[Int14] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2014. Online. Accessed 2015-03-12. `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html?wapkw=order+number+248966-025`.

[JCD+79] A. K. Jones, R. J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl. "StarOS, a Multiprocessor Operating System for the Support of Task Forces." In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pp. 117–127. 1979.

[JMG+02] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. "Cyclone: A Safe Dialect of C." In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pp. 275–288. USENIX Association, Berkeley, CA, USA, 2002.

[KAR+06] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. "K42: Building a Complete Operating System." In *Proceedings of the 1st EuroSys Conference*, pp. 133–145. 2006.

[KARH15] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris. "Shoal: Smart Allocation and Replication of Memory for Parallel Programs." In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15, pp. 263–276. Santa Clara, CA, 2015.

[KDS+13] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. "Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security." In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pp. 721–732. ACM, Berlin, Germany, 2013.

[KEH+09] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. "seL4: Formal Verification of an OS Kernel." In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pp. 207–220. ACM, Big Sky, Montana, USA, 2009.

[KKAE11] J. Kim, J. Kim, D. Ahn, and Y. I. Eom. "Page coloring synchronization for improving cache performance in virtualization environment." In *Computational Science and Its Applications-ICCSA 2011*, pp. 495–505. Springer, 2011.

[KL] D. Koester and B. Lucas. "HPC Challenge - Random Access." Online. `http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/`. Accessed 2015-03-09.

[KN93] Y. A. Khalidi and M. N. Nelson. "The Spring Virtual Memory System." *Technical Report SMLI TR-93-9*, Sun Microsystems Laboratories Inc., 1993.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[LBKN14] V. Leis, P. Boncz, A. Kemper, and T. Neumann. "Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age." In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pp. 743–754. ACM, New York, NY, USA, 2014.

[LCC+75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. "Policy/Mechanism Separation in Hydra." In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pp. 132–140. ACM, Austin, Texas, USA, 1975.

[Lina] Linux Kernel Project. "Hugetlbpage support in the Linux kernel." `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`.

[Linb] Linux Kernel Project. "Transparent Hugepage Support." `https://www.kernel.org/doc/Documentation/vm/transhuge.txt`.

[LUE+99] J. Liedtke, V. Uhlig, K. Elphinstone, T. Jaeger, and Y. Park. "How to Schedule Unlimited Memory Pinning of Untrusted Processes or Provisional Ideas About Service-Neutrality." In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, HOTOS '99, pp. 153–. IEEE Computer Society, Washington, DC, USA, 1999.

[LW09] A. Lackorzynski and A. Warg. "Taming Subsystems: Capabilities as Universal Resource Access Control in L4." In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems, Eurosys affiliated workshop*, IIES '09, pp. 25–30. ACM, Nuremburg, Germany, 2009.

[Mar12] E. Martignetti. *What Makes It Page?: The Windows 7 (x64) Virtual Memory Manager*. CreateSpace Independent Publishing Platform, 2012.

[Mil06] M. S. Miller. "Robust composition: towards a unified approach to access control and concurrency control." Ph.D. thesis, Johns Hopkins University, Baltimore, MD, USA, 2006.

[MSL+08] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. "Caja: Safe active content in Sanitized JavaScript.", 2008.

[MvRT+90] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. "Amoeba, A distributed operating system for the 1990s." *Computer*, vol. 33, no. 5, 44–53, 1990.

[MW10] A. Mettler and D. Wagner. "Class Properties for Security Review in an Object-capability Subset of Java: (Short Paper)." In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pp. 7:1–7:7. ACM, Toronto, Canada, 2010.

[Nev12] M. Nevill. *An Evaluation of Capabilities for a Multikernel*. ETH Zurich, 2012. Master's Thesis, `http://www.barrelfish.org/publications/nevill-master-capabilities.pdf`.

[NIDC02] J. Navarro, S. Iyer, P. Druschel, and A. Cox. "Practical, Transparent Operating System Support for Superpages." *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, 89–104, 2002.

[NMW02] G. C. Necula, S. McPeak, and W. Weimer. "CCured: Type-safe retrofitting of legacy code." *ACM SIGPLAN Notices*, vol. 37, no. 1, 128–139, 2002.

[NW77] R. M. Needham and R. D. Walker. "The Cambridge CAP Computer and Its Protection System." In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, pp. 1–10. ACM, New York, NY, USA, 1977.

[NZMZ09] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C." In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pp. 245–258. ACM, New York, NY, USA, 2009.

[Ora10] Oracle Corporation. Online. `http://docs.oracle.com/cd/E19683-01/806-7009/chapter2-95/index.html`, 2010. Accessed 2015-08-15.

[PFM15] T. M. Paolo Faraboschi, Kimberly Keeton and D. Milojicic. "Beyond processor-centric operating systems." In *Proceedings of the 2015 International Workshop on Hot Topics in Operating Systems (HotOS XV)*. Karthause Ittingen, Warth-Weiningen, Switzerland, 2015.

[PLZ+14] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. "Arrakis: The Operating System is the Control Plane." In *11th Symposium on Operating Systems Design and Implementation (OSDI'14)*. Broomfield, Colorado, USA, 2014.

[RAA+91] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, et al. "Overview of the chorus distributed operating systems." In *Computing Systems*. Citeseer, 1991.

[RHB+86] S. Rajunas, N. Hardy, A. Bomberger, W. Frantz, and C. Landau. "Security in KeyKOS." In *Proceedings of the 1986 IEEE Sympsium on Security and Privacy*. 1986.

[RR81] R. F. Rashid and G. G. Robertson. "Accent: A Communication Oriented Network Operating System Kernel." In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pp. 64–75. ACM, Pacific Grove, California, USA, 1981.

[RTY+88] R. Rashid, J. Tevanian, A., M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures." *Computers, IEEE Transactions on*, vol. 37, no. 8, 896–908, 1988.

[San] S. Sanfilippo. "Redis latency problems troubleshooting." `http://redis.io/topics/latency`.

[Sch17] D. Schwyn. *Hardware Configuration With Dynamically-Queried Formal Models.* ETH Zurich, 2017. Master's Thesis, `http://www.barrelfish.org/publications/ma-schwynda-hwconf.pdf`.

[Seb91] E. J. Sebes. "Overview of the architecture of Distributed Trusted Mach." In *Proceedings of the USENIX Mach Symposium*, pp. 20–22. 1991.

[SG13] D. Stolz and A. Grest. *Trace Collection, Analysis and Visualization for Barrelfish.* ETH Zurich, 2013. Distributed Systems Lab report, `http://www.barrelfish.org/publications/stolz-grest-dslab-tracing.pdf`.

[SGI14] Y. Soma, B. Gerofi, and Y. Ishikawa. "Revisiting Virtual Memory for High Performance Computing on Manycore Architectures: A Hybrid Segmentation Kernel Approach." In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '14, pp. 3:1–3:8. ACM, New York, NY, USA, 2014.

[sof] "Soft-Dirty PTEs." `https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt`.

[SSF99] J. S. Shapiro, J. M. Smith, and D. J. Farber. "EROS: A Fast Capability System." In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pp. 170–185. Charleston, South Carolina, USA, 1999.

[Tev87] A. Tevanian, Jr. "Architecture Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach." Ph.D. thesis, Pittsburgh, PA, USA, 1987. AAI8814734.

[The] The University of Tennessee. "HPC Challenge Benchmark." Online. `http://icl.cs.utk.edu/hpcc/software/view.html?id=178`. Accessed 2015-03-09.

[VBYN+14] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. "CODOMs: Protecting Software with Code-centric Memory Domains." In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pp. 469–480. IEEE Press, Minneapolis, Minnesota, USA, 2014.

[Wal] S. Wallentowitz. "Moore and More." `https://github.com/wallento/mooreandmore`.

[WLH81] W. Wulf, R. Levin, and S. Harbison. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill, New York, 1981.

[WWN+15] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization." In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pp. 20–37. IEEE Computer Society, Washington, DC, USA, 2015.

[YRSI17] P. Yosifovich, M. E. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More (7th Edition)*. Microsoft Press, Redmond, WA, USA, 7th ed., 2017.

[YWCL14] Y. Ye, R. West, Z. Cheng, and Y. Li. "COLORIS: A Dynamic Cache Partitioning System Using Page Coloring." In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pp. 381–392. Edmonton, AB, Canada, 2014.

[ZDS09] X. Zhang, S. Dwarkadas, and K. Shen. "Towards Practical Page Coloring-based Multicore Cache Management." In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pp. 89–102. Nuremberg, Germany, 2009.

[ZGKR14] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe. "Decoupling Cores, Kernels, and Operating Systems." In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pp. 17–31. Broomfield, CO, USA, 2014.