LONDON'S GLOBAL UNIVERSITY

Improving the Space Efficiency of Dynamic Memory Allocation in an Experimental Capability-Based Operating System

Szymon Duchniewicz[1]

MEng Computer Science

Supervisor: Prof. Brad Karp

Submission date: 24th May 2024

[1]**Disclaimer:** This report is submitted as part requirement for the MEng Degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

Efficient management and tracking of memory resources pose a significant challenge for computer systems due to diverse program memory requirements and system constraints. This thesis aims to improve the memory management system of CantripOS, a new experimental operating system designed for cost-constrained, memory-limited edge compute nodes, built atop the secure, formally verified seL4 microkernel. We investigate the space efficiency and performance of CantripOS's memory management system, hypothesising significant memory fragmentation due to the system's global rather than per-region memory tracking. The primary goals of this work are to improve memory bookkeeping in CantripOS, decrease failed allocation system calls, and reduce memory fragmentation. We evaluate two different memory allocators: next-fit and best-fit, and show that our improvements eliminate failed allocation system calls and substantially decrease memory fragmentation for various synthetic and representative workloads.

Contents

1 Introduction 3
1.1 Improvement of Memory Management in CantripOS . . . 4
1.2 Evaluation of Memory Management in CantripOS . . . 5
1.3 Thesis Goals . . . 5

2 Background & Related Work 7
2.1 Dynamic Storage Allocation . . . 7
2.1.1 Allocator Mechanisms . . . 8
2.1.2 Memory Fragmentation . . . 13
2.2 Physical Memory Fragmentation in seL4 and CantripOS . . . 14
2.2.1 Watermarking Constraint Fragmentation . . . 17
2.2.2 Alignment Constraint Fragmentation . . . 17
2.3 Background Information: Systems in Play . . . 18
2.3.1 CantripOS . . . 19
2.3.2 seL4 . . . 21

3 Design and Implementation 24
3.1 Original CantripOS Memory Management System . . . 24
3.1.1 Original CantripOS Memory Allocator . . . 26
3.1.2 Limitations of Memory Management in seL4/CantripOS . . . 26
3.2 Our Initial Design: Improving Memory Manager via MDB Traversal . . . 28
3.2.1 Goals . . . 28
3.2.2 What to Return From the Kernel to Improve Memory Bookkeeping? 28
3.2.3 Kernel Modifications . . . 30
3.2.4 User Space Modifications . . . 34
3.3 Our Improved Design for the CantripOS Memory Manager . . . 35
3.3.1 Goals . . . 36
3.3.2 What to Return From the Kernel to Improve Memory Bookkeeping? 36
3.3.3 Kernel Modifications . . . 36
3.3.4 User space Modifications . . . 38
3.3.5 Validation of the Designs' Correctness . . . 38
3.4 Implementing the Best-Fit Strategy for Memory Allocation . . . 39
3.5 Infrastructure for Performance Analysis . . . 40

4 Performance Evaluation 41
4.1 Experiments . . . 41
4.1.1 Measured Metrics . . . 42
4.1.2 Synthetic Workloads . . . 42
4.1.3 Representative Workloads . . . 43
4.1.4 Expected Study Outcomes . . . 48
4.2 Results . . . 49
4.2.1 Synthetic Workloads – Random with Uniform Distribution . . . 49
4.2.2 Representative Workloads – Standalone Application Traces . . . 51
4.2.3 Representative Workloads – Sequential Application Trace Interleaving 55
4.2.4 Allocation Latency Evaluation . . . 59

5 Conclusions & Future Work 61
5.1 Conclusions . . . 61
5.2 Future Work . . . 62

A Code of the Modified CantripOS and seL4 System 67
A.1 Code for all repositories used in the project . . . 67

B Memory System Redesign: Fix for dirty untyped object splitting 68
B.1 Description and bugfix of the dirty untyped object splitting bug . . . 68

C Memory System Redesign: Neighbour Traversal Inefficiency 69
C.1 Empirical confirmation of neighbour traversal inefficiency . . . 69

D Additional synthetic workload results 70
D.1 Memory statistic for 32 and 64 percentage chance for deallocation runs. . . 70

E Project Plan 73

F Interim Report 77

Acknowledgments

First and foremost, I would like to thank Professor Brad Karp, for his immense support and guidance. His expertise in conducting research and knowledge of computer systems proved to be invaluable at so many steps of the project. I am deeply grateful for being able to learn from him.

I would also like to thank Sam Leffler for building such an exciting system and sharing his knowledge on the inner-workings of CantripOS and seL4.

Further, I would like to thank Kent McLeod for explaining why the seL4 modifications proposed in this design have widely different impact on the system's performance.

Finally, I would like to thank my family for their support, especially my brothers Jakub and Michał, as well as my friends – Marek Masiak and Andrzej Szablewski, for listening to my many monologues about this project and providing constructive feedback.

Chapter 1

Introduction

Efficient management and tracking of memory resources is a complex challenge for all computer systems with no single universal solution, as programs exhibit a wide variety of memory requirements and systems impose various constraints. The centrepiece system in this work – CantripOS, is a new experimental operating system to be used in cost-constrained, memory-limited edge compute nodes. Using the limited memory efficiently is therefore essential to enabling successful execution of applications. As CantripOS aims to be a secure system but also provide a familiar interface for app developers, it is built on top of a secure, formally verified microkernel: seL4 [1]. Developing applications directly on seL4 is not straightforward due to certain complexities of the system, further explored in Section 2.3.2, hence CantripOS provides additional interfaces, such as a memory management interface, for use by developers.

As in any modern system, memory is *dynamically managed* in CantripOS, meaning the amount of memory required for running the system or its applications changes while the system is in operation. Dynamic memory allocation is orchestrated by an *allocator* – its main goals are reserving and freeing variable-size *blocks* of memory from larger regions of available memory, at the same time achieving minimal wastage of space and latency of requests.

The wasted space produced by an allocator is a result of the central problem of memory allocation – *fragmentation*. Fragmentation is categorised as *external* or *internal* [2].

External fragmentation arises when there are free blocks of memory available for use by the allocator, but cannot be used to satisfy a particular memory allocation request. For example, the requested allocation size exceeds all possible contiguous regions of memory in the free blocks.

Internal fragmentation arises when the allocator assigns a big-enough free block to satisfy a memory allocation request, but the block is larger than needed.

Space efficiency of dynamic memory allocation in CantripOS has not previously been investigated. In this work, we aim to explore this and other aspects of memory management performance, as there is reason to speculate that it suffers from severe memory fragmentation we also aim to improve it. This speculation is based on the fact that tracking memory usage in CantripOS is limited to only a global statistic, rather than an exact

per memory region statistic, leading to CantripOS "guessing" when making an allocation (Section 3.1.2 describes this in more detail). This limitation is due to the system being divided into 2 domains: the seL4 kernel domain, which tracks all memory usage and the domain of CantripOS system components, which interacts with the kernel domain via an interface similar to RPC calls[1]. It is not straightforward to share this exact memory bookkeeping information from the seL4 domain to the broader CantripOS domain.

All available memory in CantripOS is divided into a fixed number of regions, also referred to as *slabs*. These regions are determined by the underlying seL4 microkernel at boot time. Unlike in popular monolithic kernel-based operating systems such as GNU/Linux, in seL4 memory allocation for kernel objects is managed by a designated user-level application, rather than the kernel [4].

For CantripOS, this user-level application is one of the components from which the system is constructed, called the `MemoryManager`.

When CantripOS allocates memory, it does so using a *next fit* allocator mechanism – "consulting" each slab it is managing until a slab big enough to fit the requested object is found. This allocation mechanism always resumes iterating over all free slabs from the slab on which a previous allocation succeeded, the mechanism is described in more detail in Section 2.1.1. However, since the domain of CantripOS does not know how much memory is allocated and free in each individual slab (only the total of all slabs), in order to allocate a memory object it actually makes an allocation seL4 system call for each "consulted" slab. If the system call request fails, then CantripOS is aware that this slab is too small or too full to satisfy the allocation, and tries the next slab. As a result, each allocation request can result in multiple system calls, up to the total number of slabs in CantripOS – an undesirable property of any operating system.

1.1 Improvement of Memory Management in CantripOS

Our aim is to evaluate the existing memory management system and improve it, by decreasing memory fragmentation for current and future applications of CantripOS and decreasing the latency of allocation requests. Improving the CantripOS domain's bookkeeping of allocated memory to track usage at a per-slab granularity should eliminate the need to make more than one system call request for each allocation, potentially decreasing the latency of memory allocation. There are other benefits of such accurate bookkeeping: CantripOS application developers might want to know specific app's memory usage, and the CantripOS system developers would be able to assess the efficiency of general system allocator. Being able to evaluate the existing CantripOS memory allocator allows us to implement different allocator mechanisms and compare which one leads to smaller fragmenting of memory. In this work, we implement the *best fit* allocator and compare against the existing *next fit* allocator, both are described in detail in Section 2.1.1.

[1] More specifically, such cross-domain communication is handled by the *inter-process communication* mechanism: IPC. It can be thought of as an RPC mechanism, but instead of going across networks, it goes across protection-domain boundaries [3].

1.2 Evaluation of Memory Management in CantripOS

After extending the CantripOS system's memory bookkeeping to track exact memory usage per each memory region and after implementing the best fit allocator, we assess both the performance of both the original and the new allocator. Prior research tackles the performance evaluation of dynamic memory allocation with various approaches: formally bounding time and space complexity of allocators, collecting fragmentation metrics using synthetically generated memory workloads or using memory traces of real applications [5]. In order to explore the range of behaviours of both memory allocators, we use synthetic workloads, as described in Section 4.1.2. Since CantripOS is a fresh and experimental system and there are no complex applications which represent the target use cases of the system, we instead use the available sample applications to investigate the behaviour of both memory allocators under real workloads, as described in Section 4.1.3.

To perform the evaluation, we extend CantripOS with infrastructure for generating and running synthetic traces from its debug console, along with Robot Framework scripts for reproducible experiment scheduling. Further, we add infrastructure for tracing all memory requests in the CantripOS system to evaluate system behaviour when real applications are running, without the need to actually run those applications. This allows us to manipulate these traces and derive new workloads from them.

As the elements of CantripOS project are split across multiple repositories, source code for all experiments, modifications, and improvements referenced in this work as split across multiple repositories and branches, all of which are explicitly listed in Appendix A.1.

1.3 Thesis Goals

We encapsulate the main goals of this work as:

Improving bookkeeping of memory in the CantripOS domain. Extending the tracked memory usage statistic to per-slab granularity in the user space of CantripOS will allow the system's developers to assess the performance of dynamic memory allocation in CantripOS and compare different memory allocator mechanisms. It is therefore a foundational improvement which would act as a catalyst for subsequent enhancements of the memory management system.

Eliminating failed allocation system calls when allocating memory in CantripOS. With more detailed bookkeeping information in CantripOS, its memory allocator can be modified so it determines when the system is out of memory or which memory slab is big enough to satisfy an incoming memory request, instead of relying on failure/success of seL4 allocation system calls. Achieving this goal would potentially decrease the latency of memory allocation requests in CantripOS.

Reducing memory fragmentation of the memory allocator of CantripOS. The more detailed bookkeeping information in CantripOS, allows for precise measurement of memory fragmentation. We can therefore implement a different memory allocator mechanism, such as the best fit mechanism, and empirically verify whether using it successfully

reduces the fragmenting of memory in CantripOS.

Chapter 2

Background & Related Work

In this chapter, we describe the background information necessary in order to address the goals outlined in Section 1.3. We also summarise related work reviewing different techniques in dynamic memory allocation.

2.1 Dynamic Storage Allocation

One of the common factors of all computer systems is the use of memory to save input, intermediate steps or results of computational tasks. Each process running on a computer has two options for using the system's memory: statically – defining the memory required at compile time, or dynamically – requesting memory during runtime. Nowadays, it is common for processes to use both of these options.

For all kinds of systems, dynamic memory allocation is orchestrated by an *allocator* algorithm. For simplicity, a program implementing such an algorithm we will also refer to as an allocator. The main goals of an allocator are to reserve and free variable-size *blocks* of memory from a larger storage area, where these blocks should consist of consecutive memory locations [6]. Additionally, such algorithms should yield minimal wastage of space and latency of the requests [5].

When an allocator receives a request for a block of memory for which a perfectly sized free block is not found, depending on the allocator mechanism, either a larger block is split to match the size of the request,[1] or the size of the request is rounded up, and a larger block is allocated.

In the first scenario, after splitting, the remaining free memory is kept in the free list (or whatever data structure is used to represent free blocks) as a smaller block. The other, new, block of free memory (now matching in size to the request) is allocated and handed off to the requestor. However, if the oversized block was close to a perfect fit, the remaining free block might be too small to be effectively useful, leading to *external fragmentation* (Section 2.1.2, Figure 2.3) [6]. After the block is freed, if the original remaining block (or a neighbouring block) are also free, these blocks can be *coalesced* and added back as a

[1] This method is used when allocating objects in CantripOS, further detailed in Section 2.3.1.

single block to the free block data structure. Different allocators may perform coalescing differently.

In the second scenario, the full free block is removed (or marked reserved, depending on allocator method) from the free block data structure. It is handed off to the requestor, where the remaining free memory that the requestor did not need is wasted as *internal fragmentation* (Section 2.1.2, Figure 2.4). After the block is freed, it is added in its entirety (including the "fragmented" memory) back to the free block data structure. Depending on the allocator mechanism, the allocator can attempt to coalesce it with its neighbours.

2.1.1 Allocator Mechanisms

There are many widely used algorithms for dynamic storage allocation, we now explore the conventional taxonomy of allocators introduced by Wilson et al. [5], closely related to the one presented in Standish's book [7]. The majority of allocators can be divided into the following mechanism categories:

- Sequential Fit – algorithms such as first fit, next fit, best fit, worst fit;
- Segregated Free List – algorithms such as simple segregated storage, segregated fits;
- Buddy System – algorithms such as binary buddies, weighted buddies, Fibonacci buddies, double buddies;
- Indexed Fit – algorithms using more sophisticated indexing data structures to fit desired allocation policy; and
- Bitmapped Fit – specific kind of indexed fits.

Over the years, there have been many studies of different allocators, under both synthetic workloads and workloads representative of real applications. These studies led to sometimes contradictory results, showing that certain allocators significantly outperform others. It seems that certain allocators simply perform much better in specific scenarios, and there is not a single universal allocator that would perfectly fit all use cases. As the operating system considered in this work (CantripOS, Section 2.3.1) already uses a variation of an allocator with a sequential fit mechanism and the improvement we propose also uses an allocator in that category, we describe sequential fit in more detail than the other four above mechanisms. In the interest of brevity, we leave the description of indexed and bitmapped fit allocator mechanisms to [5].

Sequential Fit

Sequential fit is one of the simplest, oldest allocation mechanisms. It relies on using a single list of free memory blocks. These free lists are often doubly linked, and a boundary tag technique is used [6]. This allows for constant-time coalescing of contiguous free blocks.

On an allocation request, the free list is traversed in order to identify a free block matching the allocation policy enforced by the allocator. The worst case run time for

most sequential fit allocators therefore scales linearly with the number of free blocks. As described in Section 2.3.1, CantripOS has a fixed number of free blocks initialised at boot time, which are not split or coalesced, which makes tracking of them much simpler, and limits the maximum traversal length. In a system that splits and coalesces, you could wind up with blocks that are only a few bytes long, so number of entities in the list is going to be in the order of memory size divided by a small integer, whereas for CantripOS the number of slabs depends solely on total system memory, and does not increase during run time. Below are some commonly used sequential fit allocator mechanisms:
First fit. Searches the free list from the beginning, and chooses the first free block that is big enough to fit the requested allocation size. If the allocator iterates over the entire list without finding a suitable free block, either a new page is requested or the system is out of memory and the failure needs to be appropriately handled (e.g. it can crash). If a sufficiently large free block is found, it can either be split or allocated fully, as described in the beginning of Section 2.1. Typically, there is a threshold value of k words defined, and if the remainder of a split would result in less than k words, the block is rounded up and allocated fully, avoiding external but introducing internal fragmentation [6].

An example allocation of a 3 word block is shown in Figure 2.1, where the threshold value k is set to 0. The allocator starts at the beginning of the free list, but the first free block is too small ($1 < 3$), so it advances to the next one. The second block is large enough to fit the request ($4 \geq 3$), so it is split into two smaller blocks: a 3 word block and 1 word block. The larger block is marked as allocated and a pointer to it is returned to the requestor, and the smaller, remaining 1-word free block is kept in the free list.

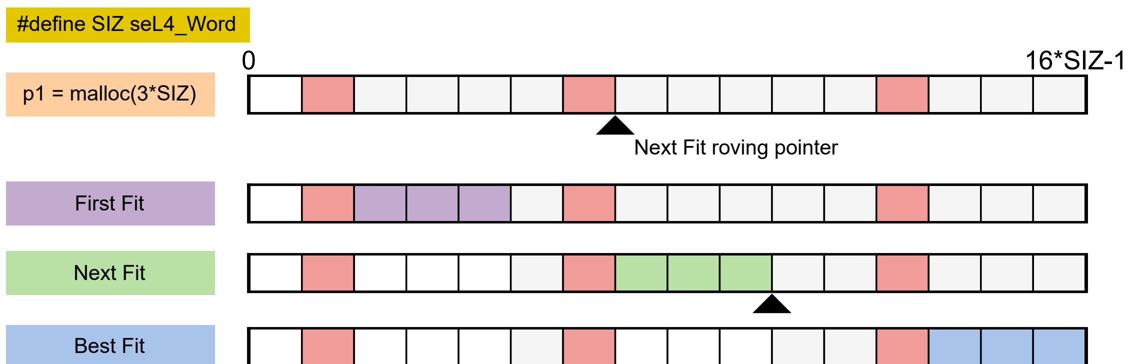


Figure 2.1: Sequential mechanisms' allocation, visualised for a 3-word allocation request. The first row shows the initial state of a memory region, along with the roving pointer position for next fit. Purple, green and blue squares indicate the position where first, next or best fit allocators would allocate those 3 words. White squares indicate contiguous, free blocks of memory, red squares are reserved/allocated blocks. Diagram inspired by [8].

Next fit. A variation of first fit, using a *roving pointer* for allocation [6]. The roving pointer tracks the address/index of the last free block from which an allocation succeeded, originally starting at the beginning of the list. Consecutive allocations begin their search for a big enough free block from the block marked by the roving pointer, wrapping back around to the start of the list if the end is reached. Similarly to first fit, when the entire

list is traversed without finding a suitable free block (roving pointer arrives at its starting location), either a new page is requested or the system is out of memory and needs to handle the failure state accordingly (e.g. crash).

An example next fit allocation of a 3-word block is shown in Figure 2.1, with the roving pointer (black arrowhead) located on the 3rd free block. This block is large enough to fit 3 words ($5 \geq 3$), so the allocator does not advance further. The 5-word free block is split into a 3-word block and a 2-word block. The larger free block is marked as allocated and a pointer to it returned to the requestor, and the smaller, remainder 2 word free block is kept in the free list. The roving pointer is adjusted to point to the now smaller, 2 word free block.

Although next fit intuitively sounds like an optimisation of first fit, empirical evidence of both synthetic and real application traces have shown that it causes more fragmentation than first fit and best fit methods. Johnstone and Wilson speculated that the better performance of first fit over next fit is due to the former method preferentially reusing blocks at one end of memory, allowing more time for the blocks on the other end of memory to coalesce. This reduces the amount of "holes" in memory. In the latter method however, this theme is much weaker [9].

Best fit. An allocator implementing the best fit mechanism, on receiving an allocation request, searches through the entire free list, in order to identify a free block that would result in the least "left over" memory if it was chosen for allocation. This "left over" memory could be either as a result of splitting, or the allocator could take into the account the amount of memory that would be wasted if the requested size was rounded up to the closest fit. The potential for best fit to leave small, unusable memory blocks spread around the memory map was recognised by Knuth [6], but Wilson et al. state that this generally does not seem to be a serious problem both for real and synthetic workloads [5].

An example allocation of a 3-word block is shown in Figure 2.1. The allocator starts at the beginning of the list, checking if the allocation would fit in the free block and how much memory would be "left over". The first free block is too small ($1 < 3$), the second and third would result in 1 ($4 - 3$) and 2 ($5 - 3$) "left over" memory words after splitting. The fourth and final block is a perfect fit, so the allocator marks it as allocated, returns a pointer to it to the requestor and removes it from the free list.

Further empirical evidence has shown that both best fit and address-ordered first fit do quite well in both random traces and real applications in terms of memory fragmentation [9], with both strategies making almost identical short-term decisions: if either allocator was used for a longer period of time, and then the other allocator was used for the next allocation, placement decisions were rarely different [10]. Wilson et al. speculate, that this shows a fundamental similarity between these two allocator algorithms, captured by an "open space preservation heuristic": "*try not to cut into relatively large unspoiled areas*" [5].

In the specific case of CantripOS however, we suspect that these similarities might not hold, given the specific limitations imposed on splitting and coalescing of blocks (explained in Section 2.3.2), resulting in frequent fragmentation of memory, as detailed in Section 2.2.

Worst cases for first, next and best fit are presented in Section 2.1.2, however, there is evidence suggesting these bounds are only very rarely reached in practice [9].

Segregated Free List

Rather than using a single list of free memory regions, the segregated free lists approach uses an array of free lists, each containing blocks of a specific size. On an allocation request, a block from a list containing matching sized blocks of memory is reserved and removed from the list. On deallocation, this block is added back to the corresponding free list. Typically, these free lists use a fixed number of size classes, and a satisfactory class match for an allocation request is a class which has an equal or slightly greater block size than the request, where no smaller class exists that could otherwise fit the requested block. A common class scheme is to use exponentially sized classes: e.g., 4 words, 8 words, 16 words, 2^k words.
Simple segregated storage. The simplest allocator using this policy, it never splits larger free blocks when a list of with a class of smaller sized blocks is empty. Instead, when the allocator is asked for a block of a size for which the free list of an appropriate size class is empty, it requests more memory from the underlying operating system. A kernel-level allocator, which manages all physical memory, simply chooses a yet-unused page. Then it splits the page into blocks of the required size class and adds them to the appropriate free list. This allocator is fast, but subject to potentially severe fragmentation.
Segregated fits. The main difference between a simple segregated storage allocator and allocators in this category is that on an allocation request for which no free block is found in the appropriate list, the allocator algorithm tries to find a larger block and split it. However, if this search fails, a new page is requested from the underlying operating system and split accordingly. Additionally, some algorithms in this category allow for free lists of blocks with varying sizes (up to some class size). This allows for a more hybrid approach between sequential fit and segregated free list mechanisms, e.g., employing a first fit or a next fit algorithm to look for a properly sized block in the smaller sized free lists.

Buddy System

The core principle of a buddy system is splitting the entire available memory in two large regions (called "buddies"), and each of those areas are further broken down into smaller and smaller regions.

It is essentially a special case of a segregated free list, and more precisely of a segregated fit with a specific constraint on splitting and coalescing. A block can only be coalesced with its "buddy" – the other region that resulted from splitting the current block's parent. A variation of this allocator mechanism – the *binary buddy system*, is used in the Linux kernel for physical page allocation [11]. It is an especially good choice for physical memory allocation, as physical memory is accessed at page granularity, so it never produces internal fragmentation (it is downstream allocators, which manipulate virtual memory mapped to

these pages that can result in their underutilisation).

Binary buddy system. As discussed by Knuth [6], this design was first used by Markowitz in the Simscript system and independently discovered by Knowlton [12]. It is the simplest variation of a buddy system: all buddy sizes are powers of two, and each buddy is split exactly into two equal blocks. An example of a 4 KiB memory region after some allocations is shown in Figure 2.2. The binary buddy system results in very easy to compute binary addressing: for a block of size 2^k, one of the buddies will have a 0 at the $k + 1$st least significant bit, while the other will have a 1. It does, however, suffer from potentially high internal fragmentation (reported as about 28% [13]), as each allocation request needs to be rounded up to the nearest (higher) power of two.

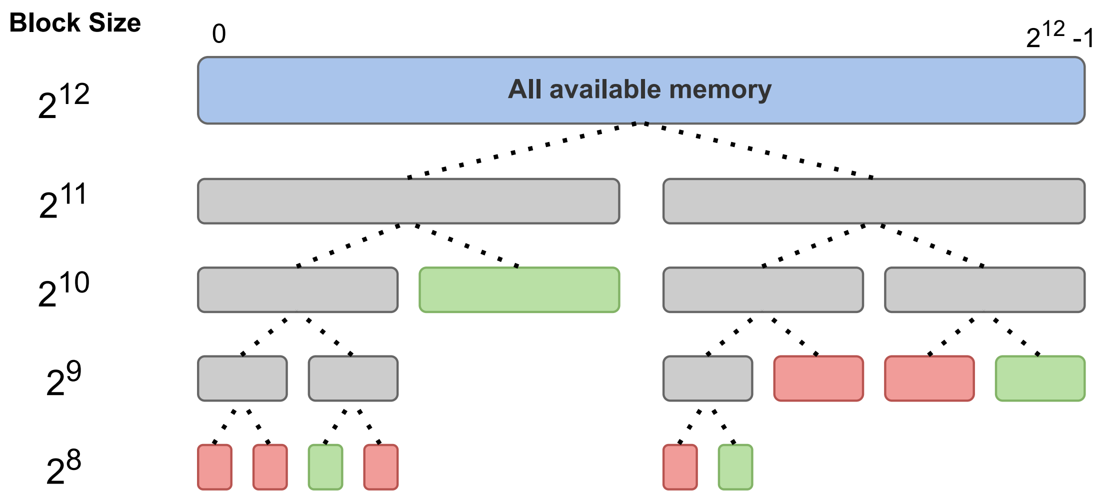


Figure 2.2: Memory map obtained with the binary buddy system. The column on the left shows the sizes of blocks at specific levels (in bytes): free (green), allocated (red), split (grey, blue). When both buddies at a level are freed, there are coalesced into the split (grey) block one level higher, at the same address.

Fibonacci buddies. As the name suggests, the main difference between this method and binary buddies is that it uses block sizes from the Fibonacci sequence [14] (or a Fibonacci-like sequence [15, 13]). This means, however, that after splitting, the resulting two buddy blocks are of different sizes, meaning that consecutive allocation requests of the same sizes will require additional block splitting. Empirical evidence using synthetic traces shows that this variation of a buddy system results in lower memory fragmentation than binary buddies for certain workloads [14, 13].

Storage (or memory), is always limited by the boundary of physical size and capacity of the electronic component (e.g. RAM, Flash storage, SSD). However, it is possible for a program to fail for lack of usable memory long before all the system's memory is used (or allocated), due to *fragmentation*.

2.1.2 Memory Fragmentation

Dynamically allocated memory is used by all sorts of applications, ranging from operating systems to smaller processes managed by OSes. Thus, it is required for allocators to serve requests for memory blocks of varying sizes, lifetimes (duration for which the reserved block is in use) and inter-arrival times [7]. It is well documented that there is no single allocator mechanism that performs optimally in all of the above described scenarios. Moreover, prior research proves that for any allocator mechanism, an adversarial workload can be constructed in such a manner to cause severe fragmentation [16, 17]. In his work, Robson [17] has shown that considering a dynamic allocation system with maximum size of a block of n words and total amount of memory allocated at any time M, the memory size necessary to overcome the effect of fragmentation lies between $\frac{1}{2}Mlog_2n$ and $0.84Mlog_2n$, for an optimal strategy. In fact, the worst-case upper bounds for the amount of memory required have been identified for some commonly implemented allocation strategies described in Section 2.1.1. For the buddy allocator mechanism [12], Knuth has shown about $2Mlog_2n$ words are sufficient. Robson further has shown that this bound is $Mlog_2n$ for first fit and Mn words for best fit [18]. However, Shore [19] showed that both first fit and best fit deal pretty well under synthetic, random workloads[2] – notably, first fit outperforms best fit for exponential distributions of allocation sizes, and best fit outperforms first fit for normal and uniform distributions.

The term *memory fragmentation* encapsulates the phenomenon of decreasing storage potential utilisation due to fragmenting (or splintering) of memory regions into many smaller areas of varying size, some of which are reserved (allocated) and unavailable for use. Fragmentation normally occurs as a side effect of using dynamic memory allocation, when memory blocks are allocated with various sizes and lifetimes. It can be categorised into two general cases: *external* or *internal* fragmentation [2].

External fragmentation occurs when there are available free blocks of memory that can't be used to satisfy an allocation request. For example, there might be no contiguous regions of memory that can satisfy a particular request, even though the total free memory is larger than the requested value. Or, this could be due to the allocator being unable to split larger blocks of memory into smaller ones (or not coalescing contiguous, free blocks of memory). An example sequence of allocation and deallocation requests handled using a first fit allocator mechanism, leading to external fragmentation, is shown in Figure 2.3. Allocations are single-word aligned. Allocation requests `p1`, `p2` and `p3` along with the free request succeed, but the following request for a 6-word block fails. There are a total of 6 free words available, in this region, but spread across 2 blocks, so it is not possible to allocate a contiguous 6-word memory region.

Internal fragmentation occurs when an allocator allocates more memory for a block than is needed, wasting the remainder. This normally occurs due to padding for alignment purposes, overhead of maintaining heap data structures (e.g. boundary tags [6]) or as a

[2] Wilson et al.[5] do point out that these workloads are insufficient to determine performance of allocator mechanisms. More detail in Section 4.1.2.

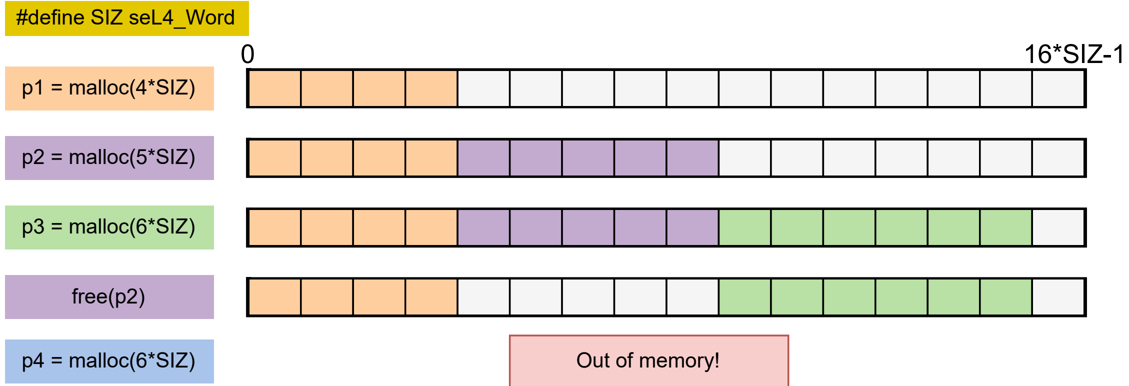


Figure 2.3: Series of allocation and deallocation requests, leading to external fragmentation. Diagram inspired by [8].

method for battling external fragmentation, by aligning blocks to specific sizes (Figure 2.4) [8, 5].

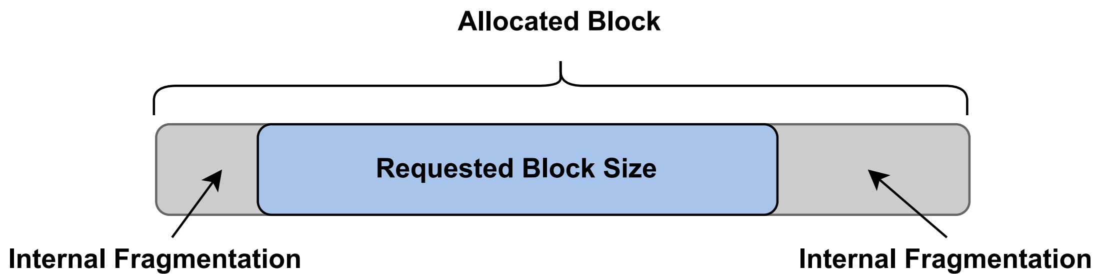


Figure 2.4: Memory lost due to internal fragmentation (gray areas) when allocated block is larger than the requested memory (blue area).

2.2 Physical Memory Fragmentation in seL4 and CantripOS

We now discuss how for dynamic memory allocation, apart from fragmentation of virtual memory, *physical* memory fragmentation is a strongly limiting factor in CantripOS. To understand how this problem emerges, it is informative to review why it is not so prevalent in a different commonly used operating system – the Linux kernel. We also highlight the distinguishing features of seL4 that make the physical memory fragmentation problem relevant. As the Linux kernel is a fast evolving software ecosystem, the reader should be aware that the following description reflects the state of kernel version 6.8.9.

Physical memory allocation in the Linux kernel. Physical page allocation is mainly handled by a binary buddy allocator in the Linux kernel [11], which always allocates memory at a fixed granularity of 1 page – meaning objects whose sizes are not multiples of the page size are not allocated. This by itself removes internal fragmentation at this level of allocation – each request is matched exactly, and internal fragmentation is introduced further down the allocation chain. Allocation of large, contiguous regions of physical pages is also fairly limited, as continuity of memory can be satisfied by allocating discontiguous physical

pages, made contiguous in virtual memory. This in turn means that there is limited external fragmentation of physical memory, in the sense that most requests for continuous memory can be satisfied by breaking them up into discontiguous physical pages.

Virtually contiguous memory allocation in Linux kernel. When dealing with large allocations, discontiguous memory physical pages are allocated using the binary buddy allocator and made contiguous in virtual memory. This is necessary because large, contiguous allocations using the buddy allocator are often hindered by external fragmentation [11]. The `vmalloc()` function is part of the API for non-contiguous memory allocation. One of its uses is to allocate pages for process' virtual memory. A diagram showing how a process (*Process A*) calling `vmalloc()` uses the buddy allocator and maps a physical page to virtual memory is visible in Figure 2.5. The diagram also illustrates a different process – *Process B*. It shows that such a virtual page mapping is not made directly to the process' page table, but is only mapped there on a page fault (when *Process B* tries to access that memory).

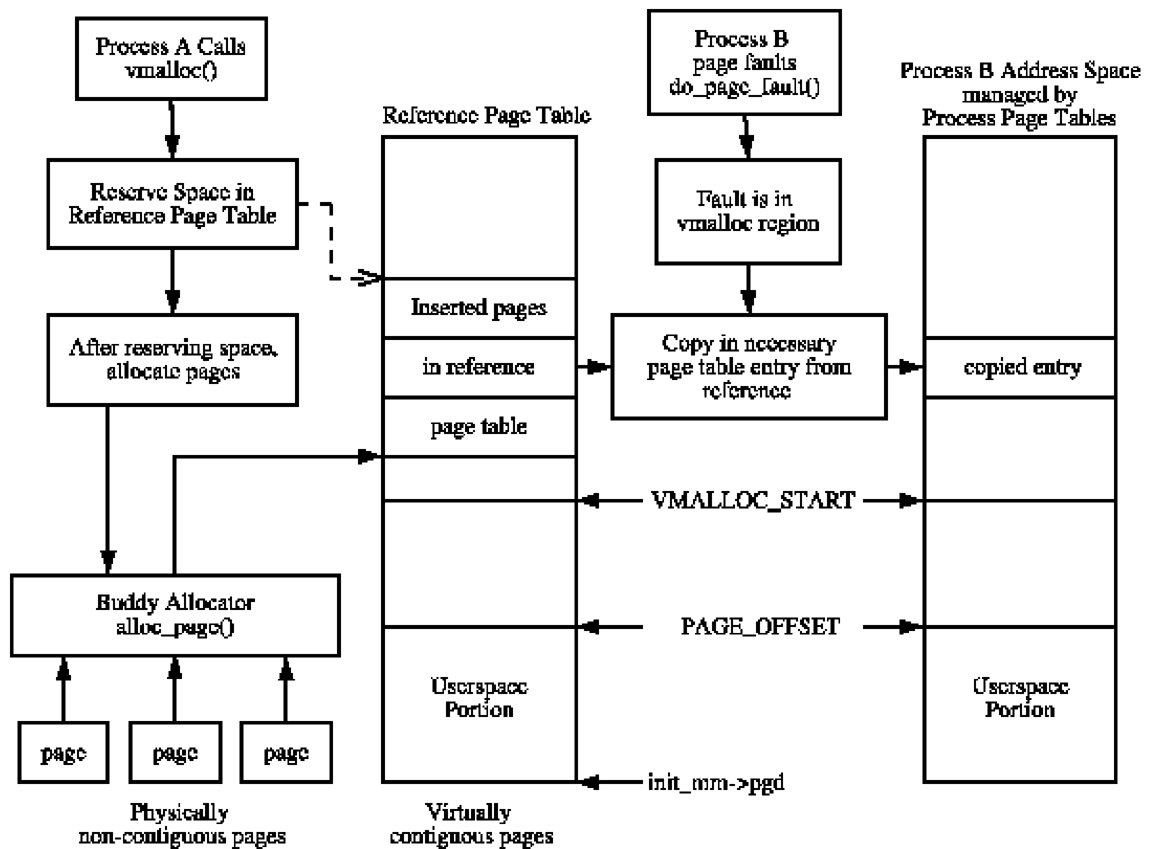


Figure 2.5: Relationship of `vmalloc()` using the physical page buddy allocator to map physically discontiguous pages to virtually contiguous. Figure source [11].

Kernel object allocation in Linux kernel. For allocating kernel objects (such as page tables, page table entries), individual physical pages are again allocated using the buddy allocator and a slub allocator [20] is used to allocate and manage individual objects. The slub allocator is an improvement to Bonwick's original Slab allocator [11, 21, 22]. Eliding detail in the interest of brevity, the main goals of slab (and slub) allocators are: using small

blocks of memory to minimise internal fragmentation that would have been introduced by using a buddy system, caching objects that are frequently used to minimise time spent on their allocation and better utilisation of hardware caches by aligning objects to the L1 and L2 caches [11].

Considering the above, when allocating memory for a process, any additional kernel objects that are allocated (e.g., page tables for virtual memory) are allocated using different allocator mechanisms and **are not mixed** in the same physical pages. The immediate result is that these allocations do not cause internal fragmentation when objects of varying sizes are used, since they reside on separate pages. Further, when a process exits, all of its memory can be immediately reused by a subsequent process.
CantripOS behaves quite differently for three reasons:

1. seL4 allows allocation of sub-page size objects in physical memory;
2. seL4 enforces object-size address alignment; and
3. seL4 enforces a watermarking (windowing) mechanism of untyped objects.

Allocation of sub-page-size objects in physical memory. In seL4, direct manipulation of physical memory allows for kernel objects to be allocated contiguously with user-space objects. This is safe (and proved to be safe [23, 24]), as accessing memory objects in seL4 is finely controlled by capabilities (described in Section 2.3.2). Different kernel objects can have different sizes and user-space objects are always multiples of a single page (for 32-bit RISC-V, a page is 4 KiB). Together with seL4's requirement of *object-size address alignment*,[3] these constraints dictate that consecutive allocations of objects with varying sizes introduce internal fragmentation in **physical** memory (demonstrated in an example request sequence in Figure 2.7).
Watermarking (windowing) mechanism of untyped objects. When the last reference to a memory-backed object is deleted in seL4, memory might not yet be freed for reuse to the memory region that held it. For each free region, the kernel keeps a *watermark* (also referred to as window edge pointer or `freeIndex` in seL4 source code) that records what portion of the region has been allocated. In seL4 terminology, free memory regions are also referred to as *untyped regions* and objects which hold this free memory are called *untyped objects*. On future object allocation requests, one of two things happens. If there are any objects still allocated in that untyped region, the watermark level is aligned to the new object size, a new object is allocated and the watermark level increases. If all objects previously allocated in that region are deleted, the kernel resets the watermark and starts allocating from the beginning of the region once more [25, Section 2.4.1].

This means that any internal fragmentation produced by *object-size address alignment* cannot be reused unless the `freeIndex` (watermark) is reset. Furthermore, any freed objects do not result in free memory until all objects in an untyped region are also removed.

[3] This is not documented in the seL4 reference manual [25], but the alignment is applied during allocation (retyping) of memory in the seL4 kernel.

Therefore, when allocating two or more threads[4] in a single untyped region, after one of the threads exits, the resulting freed memory cannot be used until all threads allocated in that region exit. This is not the case for the Linux kernel: as soon as a process exits, its memory can be reused for another process.

From the above, we conclude that physical memory fragmentation has significantly larger impact on CantripOS (and seL4) system performance than for the Linux kernel. Based on these restrictions, we define two additional subtypes of fragmentation for the purposes of this thesis: *watermarking constraint fragmentation*, and *alignment constraint fragmentation*.

2.2.1 Watermarking Constraint Fragmentation

Watermarking constraint fragmentation is a special case of internal fragmentation. It is defined as the total amount of memory at addresses lower than an *untyped object*'s watermark, that was previously occupied by objects since freed, and is not available for allocation. It is caused by the watermarking of untyped objects, described in Section 2.3.2: allocating multiple objects in a single untyped region advances the watermark by the size of the objects (and the number of bytes lost due to alignment in between those objects). Future objects can only be allocated at the address immediately after the watermark (higher). The watermark is reset to 0 only when all objects in an untyped region have been removed.

Watermarking constraint fragmentation becomes especially problematic when objects of different lifetimes get allocated in the same untyped region, as the memory will be inaccessible until all objects in that region are freed.

Figure 2.6 shows an example sequence of allocations that causes watermarking constraint fragmentation (marked as red squares in the diagram). We can observe that after freeing the object `p1` or the object `p3`, regions previously occupied by them are marked as red and temporarily lost due to watermarking constraint fragmentation. Only after the final object, `p2`, is freed does the watermark (black arrowhead) advance back to address 0 and the regions occupied by `p1, p2, p3` as well as the 2-word region between `p1` and `p2` can be used again.

2.2.2 Alignment Constraint Fragmentation

Alignment constraint fragmentation is also a type of internal fragmentation. It is defined as the total amount of memory at addresses lower than an untyped object's watermark, that was unused to meet alignment requirements. It is caused by contiguous allocation of objects with varying sizes: e.g., allocating a 2^4 byte object, such as a `seL4_ReplyObject`, followed by a 2^{12} byte allocation of a `seL4_RISCV_4K_Page` in an empty untyped region, would result in $2^{12} - 2^4 = 4080$ bytes lost to alignment constraint fragmentation.[5]

[4] seL4 does not have the concept of processes, but individual threads have separate virtual address spaces.

[5] Object sizes provided for a 32-bit RISC-V architecture [26].

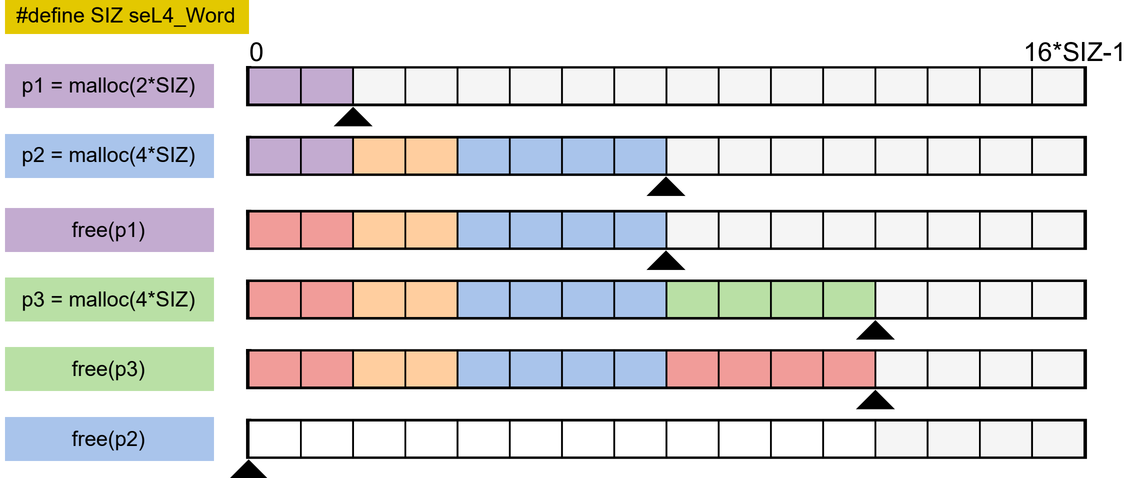


Figure 2.6: An example sequence of allocations and deallocations leading to watermarking constraint fragmentation (red squares) of an untyped region. Memory lost due to alignment (alignment constraint fragmentation) is marked as orange squares. The black arrowhead marks the positions of the watermark/`freeIndex`. Figure inspired by [8].

Figure 2.7 shows an example sequence of 3 allocations, each requesting an object of a different size. Allocating `p1` is trivial, as the watermark of the untyped object is already aligned to a multiple of the object size ($0 \mod (2 \times SIZ) = 0$). After allocating a 2 word object `p1`, the starting address of the allocation of `p2` needs to be aligned to a multiple of the size of that object: $5 \times SIZ$. This introduces 3 words (SIZ) of alignment constraint fragmentation (marked orange). Next, in order to allocate an object `p3` of size $3 \times SIZ$, the starting address is aligned to a multiple of 3 times the word-size. As `freeIndex` is currently equal to $10 \times SIZ$, the closest aligned starting address is $12 \times SIZ$, wasting 2 words of alignment constraint fragmentation.

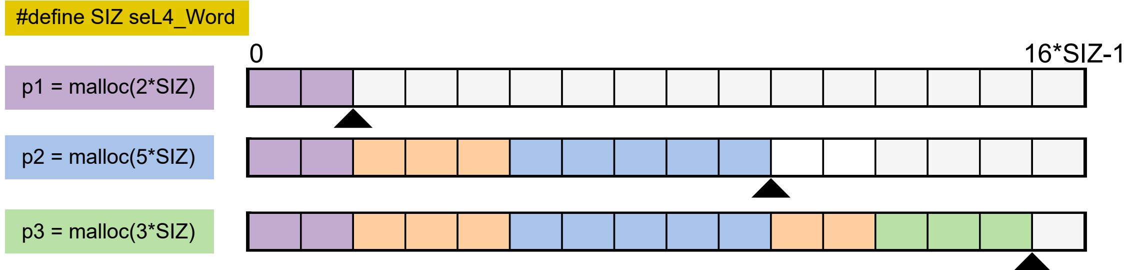


Figure 2.7: An example sequence of allocations leading to alignment constraint fragmentation (orange squares) of an untyped region. The black arrowhead marks the positions of the watermark/`freeIndex`. Figure inspired by [8].

2.3 Background Information: Systems in Play

Our work focuses on improving memory allocation in *CantripOS*, an experimental operating system for cost-constrained, memory limited edge compute nodes. CantripOS is

developed as part of Google's[6] Open Se Cura project, it aims to be a provably secure platform for building and running efficient machine learning applications [27, 28]. Designers of CantripOS decided to build the system atop the *seL4* [1] microkernel to satisfy the security requirements of the operating system, as it is the only formally verified kernel with strong guarantees of confidentiality, integrity, and availability [23, 24]. As part of this project, the team behind Open Se Cura is designing custom RISC-V-based hardware. As this hardware does not yet exist, development of CantripOS uses emulators such as Renode [29]. It emulates CPU instructions functionally, but is not clock-cycle accurate, meaning each RISC-V instruction is executed according to specification, but the time and number of emulated clock cycles it takes does not reflect a real system. This will be important when later reviewing the performance of our memory management system modifications.

We now introduce key concepts in these systems, and highlight the elements our work aims to improve – specifically the memory management system.

2.3.1 CantripOS

As seL4 is a *microkernel* (Section 2.3.2), implementations of higher-level functionalities are either not included in the kernel or are provided as components residing in user space (such as a file system, or a memory management system). This is to be expected, as seL4 by itself is not an operating system. This where CantripOS comes in: it provides services such as memory management, application management, Machine Learning model coordination, UART, and file system drivers among others.

In the domain of memory management, seL4 provides an interface for dividing and restricting access to memory regions (using *capabilities*), allocating objects (but with no allocator logic in place) and freeing objects (tracking reference count as a tree of *capability derivation history*, described in Section 2.3.2). *Capabilities* can be thought of as pointers to objects along with specific access rights attached to them, we introduce them in Section 2.3.2. CantripOS on the other hand provides abstractions for group allocation of objects, a next fit allocator handling arbitrary requests for memory, separation of static, dynamic, and device mapped memory and an interface for applications (threads) to allocate dynamic memory. All components that require dynamic memory allocation include an IPC (inter-process communication) interface with the `MemoryManger` component, and acquire it via its exposed interface.

CantripOS operating system[7] is written in the Rust programming language, and uses the memory safety features it provides through the *ownership* and *borrow checker* mechanisms [31].

It is constructed using CAmkES (Component Architecture for microkernel-based Embedded Systems) [33], an architecture for component-based development of embedded systems. The architecture of CantripOS appears in Figure 2.8, with some core compo-

[6] The system was developed in collaboration with lowRISC, Antmicro and VeriSilicon.
[7] The Rust source code for CantripOS is available in Open Se Cura repository [30].

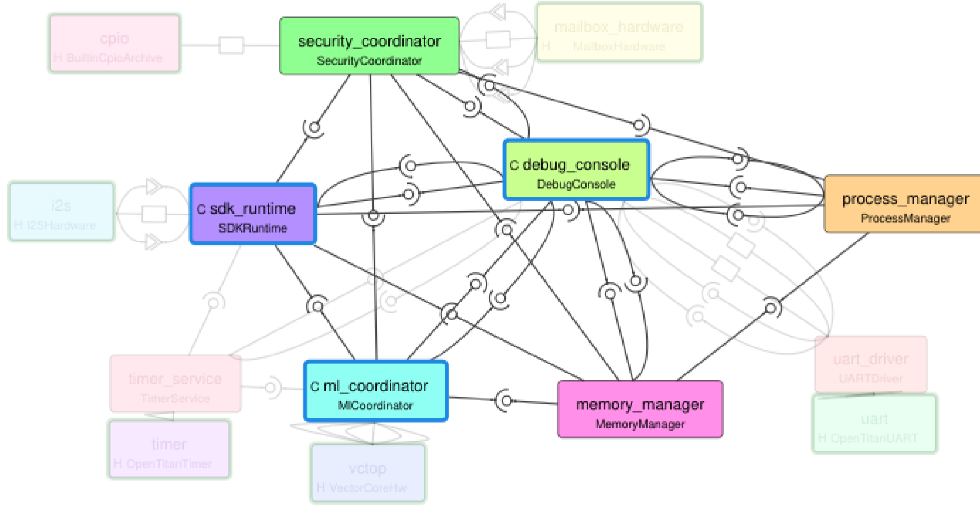


Figure 2.8: Visualisation of the CAmkES components and their interfaces for CantripOS. An edge represents an interface between two components: a curve and a circle is an IPC connection (from curve to circle), a rectangle is a dataport, a triangle is an event (here these are interrupts). Generated using a patched version of VisualCAmkES tool [32].

nents of the system highlighted for better readability. Each component is defined in a `.camkes` file and CantripOS has a single `system.camkes` for each hardware platform it supports (our project runs on the custom-built, emulated Shodan platform), which combines the individual components, defines interfaces between them and resizes the stacks of component threads where necessary. The components are created by the *rootserver* thread (application) – the first thread that gets started by the seL4 microkernel after initialisation, also known as the *root task*. It hands off capabilities to all physical, untyped (free) memory to the `MemoryManager`. The CantripOS Rust implementation of the *rootserver* application provides some extensions to the original C implementation (called *capdl-loader-app*), such as: support for reclamation of memory the *rootserver* occupied, support for CantripOS-specific CAmkES features and reduced memory consumption.

The seL4 microkernel is written in the C programming language, and some glue code is necessary in order for one to be able to interact with the other. This is mostly handled via a Rust crate (library): `sel4-sys`.[8] It provides wrappers for object passing, such as the `bootinfo` struct, which contains capabilities to untyped objects for all physical memory regions, among other things. Similarly to C system call bindings, the Rust equivalents are automatically generated from seL4 XML interface definitions, directly calling architecture specific assembly code for syscalls. For example, for RISC-V, it uses the environment call `ecall`, which can be called from user mode to request execution of a system call in kernel mode. We describe the limitations of memory management in CantripOS when presenting the original design of that component in Section 3.1.2.

[8]While working on this thesis, an official Rust crate [34] was in development by contributors in the seL4 Foundation, however the Open Se Cura project began before its creation and maintains a slightly different interface.

2.3.2 seL4

The kernel of CantripOS is the Secure Embedded L4 (seL4[9]) operating system microkernel. It is an open-source project released in 2009 by the Trustworthy Systems group at UNSW Sydney (previously CSIRO Data61) belonging to the L4 microkernel family [36]. At the time of writing this thesis, seL4 is the only operating system kernel with formal verification of functional correctness and proofs of enforcing security properties of confidentiality, integrity, and availability [23, 24, 1].

An *operating system* (OS) is the low-level system software that controls a computer system's resources and enforces security [1]. An OS can operate in a privileged (kernel) mode, giving it direct access to hardware and more elevated execution of the CPU, than applications which run in user mode. Figure 2.9 shows an abstracted view of a larger

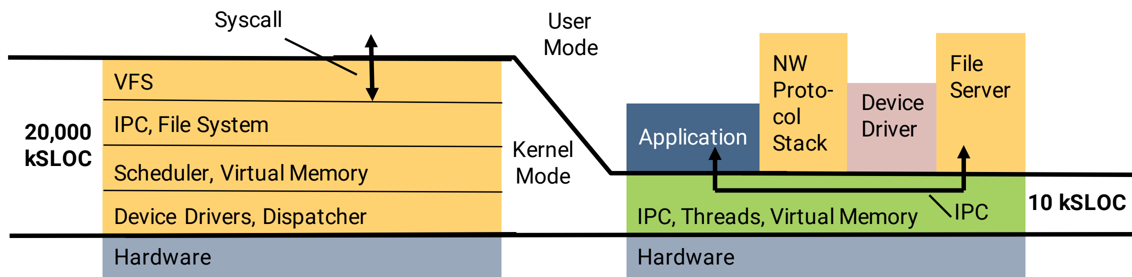


Figure 2.9: Operating-system structure: Monolithic kernel (left) vs microkernel (right). Figure source [1].

monolithic kernel, such as Linux (on the left) and a microkernel (on the right). Unlike in a monolithic kernel, a lot of components of a microkernel are executed outside of kernel mode. One of the motivations behind designing microkernels is to decrease the *trusted computing base* (TCB), defined as the subset of the overall system that must be trusted to operate correctly for the system to be secure. This by itself drastically reduces the kernel's attack surface: seL4's TCB is of the order of ten thousand lines of source code, while the size of the Linux kernel is of the order of 20 million lines of source code [1]. The downsides of microkernel architecture is that the kernel provides very minimal services: just enough to securely multiplex hardware resources and isolate threads, along with *inter-process communication* (IPC) protocol for communication and remote procedure execution between threads. In order to provide the same services a monolithic OS implements in the kernel, systems built with microkernels implement these services as programs (just like user-space applications), each running in their own sandbox, with an IPC interface for those programs to call [1].

In seL4, apart from a small amount of static kernel memory, all physical memory is managed by user level. *Capabilities* to all objects created at boot time along with the rest of physical memory seL4 can access are passed to the *root task*. In seL4, free memory is referred to as*untyped* memory. Untyped capabilities are capabilities to untyped memory.

[9]Note: seL4 is pronounced "ess-e-ell-four". The pronunciation "sell-four" is deprecated [1]. C source code is maintained on GitHub [35].

They can be later *retyped* into kernel objects and *frame objects* along with capabilities to them, or other (smaller) untyped capabilities. *Frame objects* are physical memory frames, which can be mapped into virtual memory and are accessible to user-space applications. Retyping can be thought of as allocating memory: once a portion of an untyped capability is retyped to a kernel object, that kernel object is created and pointed to by the newly created capability. The seL4 kernel keeps track of a *watermark* for each untyped capability. To satisfy the correctness proofs of seL4, the system only allocates objects at addresses greater than the watermark [23, Section 3.2]. After each allocation, seL4 advances the watermark to point immediately after the just-allocated object. The only scenario of the watermark moving backwards is when capabilities to all objects in a particular untyped object are deleted. The watermark is then reset to 0. We later refer to such an event as a *slab reset*. This constraint on untyped objects is the sole reason behind watermarking constraint fragmentation, Section 2.2.1.

Capabilities in seL4

A capability is an immutable tuple of an object reference and access rights, as shown in Figure 2.10. For brevity, we provide a short introduction to capabilities, leaving the exact details of capability benefits to [1]. In seL4, the only way to perform any operation on an object is by *invoking* its capability. Access right granularity is far greater for a capability based system, than for example a system that relies on access-control lists, such as Linux. In seL4, are capabilities are stored in special objects: *capability nodes* (CNodes). A CNode is a table of slots, where each slot may contain further CNode capabilities.

Figure 2.10: A capability is a unique token which contains specific rights to a particular object. Figure source [1].

Relevant seL4 Object Invocations

In seL4, any operations on objects are performed via object *invocations* – a set of functions that can be executed when given a capability with authorised access rights. The memory management system of CantripOS relies on two object invocations, in particular: `seL4_Untyped_Retype` for allocating memory and `seL4_CNode_Delete` for freeing memory.

`seL4_Untyped_Retype` changes the type for a fragment of untyped memory held by an

untyped object into another seL4 type. This object invocation must be called on a capability for an untyped object. Returns 0 upon success, otherwise returns an error code. It is worth noting that although the user space can request to allocate all types of objects (including kernel objects), the only objects accessible by it are called *frame objects*. These can be mapped into a page table and userland programs can write to their memory through the virtual memory abstraction. Whenever untyped objects are retyped, a special data structure, called *capability derivation tree*, is updated by adding the newly created object to the hierarchy of the parent untyped object. In the C implementation of the seL4 microkernel, this structure is represented as a doubly-linked list, equivalent to a prefix traversal of the capability derivation tree. It is called the *Mapping Database* (MDB).

`seL4_CNode_Delete` deletes a specified capability and, if the last reference to an object was deleted, frees the underlying memory. Returns 0 upon success, otherwise returns an error code. It is worth noting, however, that it does not return any information on whether this was the last reference to an object (memory freed) and where the memory was freed. We describe in Section 3.1.2 the limitations of memory management in CantripOS that result from this.

Chapter 3

Design and Implementation

3.1 Original CantripOS Memory Management System

The original CantripOS memory manager, implemented in the `MemoryManager` component, runs in a separate thread initialised by the `rootserver`. After system bootstrap, the `rootserver` hands over all capabilities to untyped objects in the system to the `MemoryManager`, including those that already are partially occupied by other system component instances, as well as the `rootserver` objects.

The memory manager deletes all capabilities used in the `rootserver` by calling `seL4_CNode_Delete` on the top level CNode constructing the `rootserver`. As the memory manager holds capabilities to all available physical memory, this includes the untyped capability that holds the objects for all CantripOS components. As these components occupy a fixed amount of memory and are never removed, if the untyped capability holding them were used for dynamic memory allocation, memory in that capability would never be freed because of the watermarking mechanism described in Section 2.2.1. In order to reuse that memory, the untyped object is split into smaller objects beyond the current watermark index, which at `MemoryManager` initialisation points to the end of the memory region occupied by the system components. To match alignment and minimise waste, the `MemoryManager` creates a few smaller untyped objects. During the development of our improvements to the memory manager, we discovered that the algorithm for splitting the "dirty" untyped object had a bug that was only triggered when system memory use grew larger than half the size of the untyped object. This issue, along with a fix, is described in more detail in Appendix B.1.

Afterwards, all untyped objects are separated into 3 categories: *static*, *device* and *untyped*. *Static* untyped objects are used for allocating static memory, which is not expected ever to be freed by the system. There is very little use of it currently in the system. At the time of writing this thesis, *device* untyped objects are not used in CantripOS. Finally, the last category, *untyped*, includes all untyped memory that is used for dynamic allocation of every seL4 object.

Each untyped object in each category is wrapped in an `UntypedSlab` struct. Here, "slab" refers to a contiguous region of physical memory. Allocations from those slabs

"break off" pieces of contiguous physical memory, retyping them into the requested seL4 object types and advancing the watermark, as described in Section 2.3.2. Each `UntypedSlab` struct keeps track of:

- `cptr` – index into the `MemoryManager` thread's CNode containing the capability for the untyped object,
- `_size` – size (represented as bit-width[1]) of the untyped region pointed to by the untyped capability,
- `free_bytes` – number of bytes free for allocation in the slab,
- `_base_paddr` and `_last_paddr` – base and top physical addresses of the untyped region covered by the untyped capability.

The original implementation did not precisely track the number of free bytes per slab, though it was initialised to the correct value at system start. We explain this limitation of the original design in more detail in Section 3.1.2. Also, `_size`, `_base_paddr` and `_top_paddr` are not set to the actual, correct values for the slabs created during the splitting of "dirty" untyped objects, as described above. These attributes were not used in the original design, but had been added as placeholders for future improvements.

All `UntypedSlab`s are grouped into lists and sorted in descending order by size of the untyped regions covered by each untyped object, determined by the `free_bytes` attribute of each slab. For performance purposes, each list is represented as a Rust `SmallVec` [37], statically allocated on the `MemoryManager` thread's stack. The lists are: `_device_untypeds`, `static_untypeds` and `untypeds`.

The basic interface of the `MemoryManager` component exposes four functions: `alloc()`, `free()` and two functions for getting memory statistics. Both `alloc()` and `free()` accept `ObjDescBundle` structs, which contain a collection of individual object descriptors, represented by `ObjDesc` structs. Additionally, `alloc()` accepts a `MemoryLifetime` enum, which indicates whether an allocation is static or dynamic. However, the system currently does not use any static allocations after boot. Their signatures are shown in Listing 3.1.

```
...
pub trait MemoryManagerInterface {
    fn alloc(
        &mut self,
        bundle: &ObjDescBundle,
        lifetime: MemoryLifetime,
    ) -> Result<(), MemoryManagerError>;
    fn free(&mut self, bundle: &ObjDescBundle) -> Result<(),
   MemoryManagerError>;
    ...
}
...
```

Listing 3.1: `MemoryManager`'s basic interface for memory management.

[1] Bit-width is the number of bits used for the underlying binary representation. In seL4, all object sizes are always represented in terms of bit-width.

Additionally, the memory manager exposes a collection of interface functions for use by other components of the system. This interface is compiled with each component that needs to dynamically allocate memory and handles the IPC communication with the `MemoryManager` thread to make allocation and deallocation requests.

3.1.1 Original CantripOS Memory Allocator

The memory allocator uses the *next fit* allocation mechanism, as described in Section 2.1.1. The roving pointer is tracked as an index into the `untypeds` vector, and stored in a `cur_untyped` variable.

When an allocation request arrives, and `alloc()` is called within the `MemoryManager`, it attempts to allocate an object for each object descriptor present in the `ObjDescBundle` function parameter. It does so by iterating over every slab in the `untypeds` vector, starting from the one pointed to by the roving pointer. It attempts to allocate the object on that slab by invoking `seL4_Untyped_Retype`. On success, it updates the global memory bookkeeping information, such as total allocated object count and total allocated byte count, and continues to the next object in the request list. On failure, the `cur_untyped` roving pointer is incremented, wrapping back to the beginning of the list when it exceeds the total number of elements in the `untypeds` vector. `MemoryManager` records the failed allocation attempt. If the roving pointer arrives back at its starting position for a single allocation, this means that no slab is big enough to fit the required object, and the function returns an error, and records that an out of memory (OOM) occurred.

Freeing memory is much simpler on the userland side: `free()` is called within `MemoryManager` and iterates over each object descriptor within the `ObjDescBundle` function parameter. For each object descriptor, the `delete_caps()` function is called, which in turn invokes `seL4_CNode_Delete` for each object in the CNode pointed to by the object descriptor.[2] On success, the global memory bookkeeping information is updated by decreasing total allocated object count and total allocated byte count. However, there is a possibility that the number of objects tracked in CantripOS differs from the actual count if the capability deleted is **not** the final capability pointing to that object. This is an immediate effect of CantripOS's not doing reference counting in the user space, further explained in Section 3.1.2. On failure, nothing happens.

3.1.2 Limitations of Memory Management in seL4/CantripOS

The CantripOS memory manager's bookkeeping of allocated memory is only at the global system level **in user space**: it does not track per-`UntypedSlab` statistics for the number of allocated objects, free bytes and the current position of the watermark. But this information is important, because after booting, all memory is managed in user space, not the kernel, as described in Section 2.3.2. Without this information, each allocation

[2] For memory efficiency, object descriptors for objects of the same type and size that are contiguous in a CNode are often merged together into a single descriptor, increasing the count of objects referred to by that descriptor. The `delete_caps()` function then iterates over each object in the object descriptor, calling the delete invocation for each one.

request requires "guessing" whether a slab has enough space to fit an object, or sending an allocation IPC request to the seL4 microkernel for every slab. This design does not allow introducing an allocator more sophisticated than next fit or first fit. Two limitations of the current implementation of memory management in CantripOS prevent keeping more detailed statistics about memory, and thus possibly lead to incorrect global statistics: *lacking information about successful object deletion* and *lacking information about freed memory's location*.

Lacking information about successful object deletion

Currently, when freeing an object, there is a possibility of an integer underflow, temporarily avoided by explicitly checking for underflows on subtraction. A capability in seL4 can be copied or badged (a copy of a capability with fewer rights), and `seL4_CNode_Delete` call used for deleting capabilities only returns whether the request succeeded or failed. Section 2.3.2 describes this limitation in more detail, and explains how objects are reference-counted in seL4. When a free request arrives, it is possible that the object was not yet freed, but that only a capability to that object was removed. Thus, if on each free request the global memory statistics are updated, an incorrect global allocated object count would result when two free requests arrive for two distinct capabilities pointing to the same object.

Lacking information about freed memory's location

After memory is successfully freed (assuming the deleted capability was the final capability pointing to the object backed by that memory), CantripOS only receives a boolean `true` confirmation from the seL4 kernel. As CantripOS does not track exactly from which `UntypedSlab` the memory was originally allocated, it has no way of knowing which untyped object this memory is being returned to, and whether it was the last object in the slab, which would cause the underlying untyped object's watermark pointer to be reset.

To summarise, these two additional pieces of information are necessary to improve the precision of bookkeeping of allocated memory globally, as well as to narrow the bookkeeping to per-slab granularity. The following sections describe how we extended the seL4 microkernel to provide this information, as well as how we extended the user space of CantripOS to track this information.

It is also worth considering why we rejected an alternative approach that would not require any modifications to the seL4 kernel. This approach was advised against by Sam Leffler, the core developer of CantripOS, who said it would be a complex modification requiring changes to multiple system components, and thus not worth the effort [38]. This purely user-space approach can be described in two steps:

1. Update the object descriptor structure definition to include a reference to the `UntypedSlab` in which the object is allocated. This solves *lacking information about freed memory's location*, as the location can be deduced from the object descriptor on a successful `free()` call.

2. Update the `MemoryManager` to reference-count each object descriptor. This solves *lacking information about successful object deletion*, as the global/per-slab memory statistics must only be updated once a final reference to an object is deleted.

This seemingly sound solution brings several drawbacks, which ultimately led to rejecting this alternative design:

1. Each object descriptor takes up more memory, and there can be many objects allocated, even in an embedded system.

2. The object descriptors can no longer be merged together in order to save space and potentially decrease the number of IPC calls when freeing a larger application. This effect can also have a knock-on impact on the system's extensibility, if the layout of the untyped memory hierarchy is modified in CantripOS.

3. This alternative design ends up replicating memory bookkeeping already present in the seL4 microkernel: it reproduces ref-counting of objects and maintaining the derivation history of objects, all already managed by the Mapping Database in seL4 (Section 2.3.2).

3.2 Our Initial Design: Improving Memory Manager via MDB Traversal

The lack of per-slab statistics on allocated memory in CantripOS before our modification prevents CantripOS from using more efficient allocation strategies.

3.2.1 Goals

1. To be able to allocate memory with a best-fit allocation strategy.

2. To be able to do bookkeeping of allocated memory accurately on a per-slab basis.

3. To be able to return more metadata than just an error code on every memory free call (`seL4_CNode_Delete()`). More precisely: to return the ID of the slab to which memory was returned.

3.2.2 What to Return From the Kernel to Improve Memory Bookkeeping?

The entire interface for memory allocation and deallocation relies on the Object Invocations `seL4_Untyped_Retype()` and `seL4_CNode_Delete()`. The `seL4_Untyped_Retype()` invocation is used for allocating new objects, while `seL4_CNode_Delete()` deletes capabilities, including ones pointing to objects, and is used to free memory. Full descriptions of both invocations appear in Section 2.3.2.

One method of improving memory bookkeeping in CantripOS is to keep track of memory statistics per untyped slab. To do so, the `MemoryManager` component must know 3 things from the kernel:

1. On allocation – whether it succeeds or not.

2. On free – whether memory is freed or not. It is not enough just to know when a capability is deleted. One must also know whether the capability was pointing to a physical object, and whether this was the **last** capability pointing to that object.

3. On free – which untyped object (and therefore `UntypedSlab`) the memory is returned to.

The rest of the required information can be tracked in user space. Here is a design for the bookkeeping:

At boot time/MemoryManager initialisation:

Create an array called `untypeds` with a metadata object for each slab, keeping track of:

- **Total space in the slab**: this is returned from the kernel in the `bootinfo` structure, and recalculated from those values when splitting untyped objects.

- **Allocated bytes in the slab**: how many bytes are currently unavailable for allocation.

- **Number of objects in the slab**: how many objects are currently allocated.

During allocation

1. Based on the chosen allocation strategy, select the slab for allocating an object.

2. On success returned from the kernel, update the slab metadata element in the `untypeds` vector by incrementing the **number of objects** attribute by the total count of newly allocated objects and increment the **allocated bytes** attribute by the total size of allocation, including memory wasted due to alignment (see Section 2.2.2 for more information on alignment).

3. On failure, proceed according to the chosen allocation strategy(e.g., in the original CantripOS, try the next slab).

During free/deallocation

1. On success returned from the kernel, also capture a boolean indicating whether any memory was actually freed and the index of the slab to which the memory was returned. The boolean is necessary, as objects are reference counted in the kernel and not all capabilities point to objects backed by physical memory (see Section 2.3.2 for more information on ref counting).

2. Update the slab metadata element in the `untypeds` array with the index returned from the kernel, decrementing the **number of objects** by the count of objects freed. If the **number of objects** drops to 0, the slab is reset and the attribute **allocated bytes** of the slab metadata is set to 0 (see Section 2.2.1 for more information on slab resetting). This latter mechanism tracks seL4's watermarked behaviour.

Of the three pieces of information needed from the kernel, the original memory manager implementation only receives the first one: whether an allocation succeeded. Therefore, the kernel code for the `seL4_CNode_Delete` Object Invocation needs to be modified to return the index of the slab to which memory is returned and whether any memory has actually been freed. We now describe the corresponding kernel modifications.

3.2.3 Kernel Modifications

The sole kernel object invocation that needs to change is `seL4_CNode_Delete`. The modification can be broken down into two steps: *returning two additional values from the invocation* and *getting the correct values*, a boolean and an unsigned integer.

Returning two additional values from seL4_CNode_Delete

We extend the seL4 Object Invocation XML interface definition for `seL4_CNode_Delete` to include two additional return parameters, with the direction (`dir`) attribute set to `out`. This is similar to returning of multiple values already implemented for `seL4_Untyped_Describe`.[3]

The first parameter, `untypedSlabIndex`, is of type `seL4_Word`. It contains the index of the CSlot in the `MemoryManager` thread's topmost CNode, containing the capability of the untyped object to which memory is being returned. If no memory is returned or the slab cannot be identified, 0 is returned. By convention, the zeroth CSlot in a CNode is kept empty, for the same reason as keeping the zeroth Page Table Entry unmapped in a process's virtual memory: to avoid errors when uninitialised slots are used unintentionally.[4] Therefore, there is no risk of the 0 being misinterpreted as a valid CSlot.

The second parameter, `isLastReference`, is of type `seL4_Bool`. If it is true, it indicates that the deleted capability points to an object backed by physical memory, and it is the last capability pointing to that object in the Mapping Database (MDB). When it is false, then there is at least one more capability pointing to the same object, or it is not a physical capability. In the second case, there is no requirement to find the correct value for `untypedSlabIndex` attribute, and 0 can be immediately returned in its place.

The modified XML definition of `seL4_CNode_Delete` interface, including the two added parameters, is shown in Listing 3.2.

```
1 <!-- Omitted for brevity --!>
2 <method id="CNodeDelete" name="Delete" manual_name="Delete" manual_label="
    cnode_delete">
3     <brief>
```

[3]The XML definition can be found on the Open Se Cura repo [39].
[4]Source [40].

```xml
        Delete a capability
    </brief>
    <description>
        <docref>See <autoref label="sec:cnode-ops"/>.</docref>
    </description>
    <cap_param append_description="CPTR to the CNode at the root of the
    CSpace where the capability will be found. Must be at a depth
    equivalent to the wordsize."/>
    <param dir="in" name="index" type="seL4_Word" description="CPTR to the
    capability. Resolved from the root of the _service parameter."/>
    <param dir="in" name="depth" type="seL4_Uint8" description="Number of
    bits of index to resolve to find the capability being operated on."/>
    <!-- Modified content start --!>
    <param dir="out" name="untypedSlabIndex" type="seL4_Word" description="
    Index of the Untyped in the top level CNode to which memory is returned
    ."/>
    <param dir="out" name="isLastReference" type="seL4_Bool" description="
    True if this operation deletes the last capability pointing to an
    object."/>
    <!-- Modified content end --!>
    </method>
<!-- Omitted for brevity --!>
```

Listing 3.2: First design: Modified XML of the seL4_CNode_Delete Object Invocation interface.

Modifying the interface means the automatically generated Rust stub now returns a Rust structure which contains two additional fields: untypedSlabIndex and isLastReference. All functions in user space which use this invocation need to be updated to properly unwrap and handle these fields and error code. In most cases, the two additional returned values can be safely ignored, apart from in the delete_caps() function, implemented for the MemoryManager struct, where the value is fully unwrapped and used. This functionality is described in more detail in Section 3.2.4.

Finally, the kernel function invoked by seL4_CNode_Delete needs to be modified. We extend invokeCNodeDelete[5] to additionally take a word_t *buffer parameter. This parameter is necessary to satisfy the function signature of the setMR function, but it would only be used to pass the values untypedSlabIndex and isLastReference if there weren't enough Message Registers. On RISC-V cores, seL4 configures registers a2, a3, a4 and a5 as message registers.

These values are set in the 0th and 1st message registers (the a2 and a3 registers on RISC-V), using the setMR helper function. As shown in Listing 3.3, these values are only modified if no exception occurred while capability deletion took place inside the cteDelete() function. This is because if, for example, a pre-emption occurs, the message registers used for function input (which are reused for output) would be overwritten by our new untypedSlabIndex and isLastReference values. After returning on line 44, these values would be stored by the calling function, assuming they are the input param-

[5]The invokeCNodeDelete function is defined in the seL4 C implementation [35].

eters to `invokeCNodeDelete`. When pre-emption occurs and registers are restored for the `invokeCNodeDelete` function, it would restart with the wrong input parameters.

```c
exception_t invokeCNodeDelete(cte_t *destSlot, word_t *buffer)
{
    exception_t status;
    word_t untypedSlabIndex = 0;
    bool_t isLastReference = false;
    word_t MEMORY_THREAD_CNODE_CPTR = 1;
    // Extract the required book keeping values and assign to untypedSlabIndex and isLastReference
    if ((isLastReference = isFinalCapability(destSlot))) {
        cte_t *parent_slot = destSlot;
        while (mdb_node_get_mdbPrev(parent_slot->cteMDBNode)) {
            parent_slot = CTE_PTR(mdb_node_get_mdbPrev(parent_slot->cteMDBNode));
            if (cap_get_capType(parent_slot->cap) == cap_untyped_cap) {
                break;
            }
        }
        // If found the source untyped cap, get its index in the memory manager thread's TCB's uppermost CNode
        if (cap_get_capType(parent_slot->cap) == cap_untyped_cap) {
            lookupCapAndSlot_ret_t lu_ret_cnode = lookupCapAndSlot(NODE_STATE(ksCurThread), MEMORY_THREAD_CNODE_CPTR);
            if (cap_get_capType(lu_ret_cnode.cap) == cap_cnode_cap) {
                cte_t *cnode = CTE_PTR(cap_cnode_cap_get_capCNodePtr(lu_ret_cnode.cap));
                word_t radix = cap_cnode_cap_get_capCNodeRadix(lu_ret_cnode.cap);
                // Find index of our target untyped in the CNode
                word_t max_slot = (1<<radix) -1;
                for (word_t slot = 0; slot <= max_slot; ++slot) {
                    cte_t *query = &cnode[slot];
                    if (cap_get_capType(query->cap) == cap_null_cap) {
                        continue;
                    }
                    if (query->cap.words[0] == parent_slot->cap.words[0] && query->cap.words[1] == parent_slot->cap.words[1]) {
                        untypedSlabIndex = slot;
                        break;
                    }
                }
            }
        }
    }
    status = cteDelete(destSlot, true);
    // Only change MRs if status is EXCEPTION_NONE (otherwise if call is preempted, input registers will be dirited)
    if (status == EXCEPTION_NONE) {
        setMR(NODE_STATE(ksCurThread), buffer, 0, untypedSlabIndex);
        setMR(NODE_STATE(ksCurThread), buffer, 1, isLastReference);
```

```
42    }
43    return status;
44 }
```

Listing 3.3: Second design: modified `invokeCNodeDelete` function in seL4 kernel.

Getting the correct values for `untypedSlabIndex` and `isLastReference`

To get the correct value for `isLastReference`, the kernel needs to check whether the capability being deleted is backed by physical memory and if it is the last reference to the pointed-to object. If so, this results in the underlying object being deleted and memory being returned to its original untyped slab. Whether a capability is the last reference to an object can be checked by looking at the nearby nodes of the Mapping Database (MDB).

As a reminder, MDB is a doubly-linked list. Traversing it is equivalent to a prefix traversal of the capability derivation tree. In seL4, whenever a new capability is created pointing to the same object, `invokeCNodeInsert` is also called, inserting a new MDB node next to the newly created object capability's MDB node. The MDB is discussed in more detail in Section 2.3.2.

Because of the above, MDB nodes of capabilities pointing to the same object are always grouped together, so it is sufficient to check whether the capabilities of the preceding and succeeding MDB nodes point to the same object as the capability being deleted. If there are no neighbouring MDB nodes whose capabilities point to the same object, then this is the last reference to that object.

This logic is handled by the `isFinalCapability()` function, present in the kernel. This is done on line 8 of Listing 3.3.

Getting the value for `untypedSlabIndex` is slightly more complicated, but can be entirely avoided if `isLastReference` is false. That is why the boolean value is checked first, and `untypedSlabIndex` is set to value 0 if the boolean is false.

If indeed the capability being deleted is the last reference to a target object, then the MDB doubly-linked list can once again be used to determine which untyped object (memory slab) the target object came from. We implement this with a backwards traversal using a while loop, with the helper function `mdb_node_get_mdbPrev()` which returns the next-higher MDB node in the capability derivation hierarchy. The traversal is shown on lines $9-14$ of Listing 3.3. The traversal stops at the first untyped capability, rather than the last. This is because, for example, the untyped slab reclaimed by the `MemoryManager` from the `rootserver` thread's memory is partially allocated and is split into a few smaller slabs, so objects located on those slabs will have more untyped objects in their derivation hierarchy.

If during the traversal the untyped capability is not found then the capability that is being deleted is not a physical capability and `untypedSlabIndex` can be safely set to 0. After the capability to the untyped slab is found, we perform a linear search through all CSlots of the `MemoryManager` thread's top-level CNode, in which we compare each non-null capability, in lines $18-36$ of Listing 3.3. There are existing functions for comparing objects

and memory regions within the kernel: `sameObjectAs` and `sameRegionAs`. Unfortunately, `sameObjectAs` only works for capabilities that are not untyped, and `sameRegionAs` would return incorrect values (e.g., the untyped slabs derived from the reclaimed memory are all within the same memory region). However, capabilities are defined as unforgeable and unique tokens in the seL4 specification [25, Section 2.1] and the seL4 capability tutorial[6], hence we compare them directly. If the capability of the parent untyped object and a capability in the top-level CNode match, `untypedSlabIndex` must be assigned the index of the CSlot containing the matching capability.

3.2.4 User Space Modifications

Apart from the functions that use the `seL4_CNode_Delete` object invocation but don't require the newly introduced return values, the only file that needs to be modified is the `MemoryManager` component's `mod.rs` file. The files, which don't require the additional output, are modified to unwrap the returned error code and ignore the rest.

Two things change in the `mod.rs` file: First, the `UntypedSlab` structure definition is expanded by two additional fields: `allocated_bytes` and `allocated_objects`. Second, the `alloc()` and `free()` functions are modified to update the memory bookkeeping values for each slab.

With the following modifications made, new allocation strategies can be implemented, better logging can be put in place and new benchmarks can be established for measuring the space and computational efficiency of this memory management system. We further explore these ideas in Sections 3.4 & 4.

Expanding UntypedSlab definition

There are two fields necessary to keep per-slab memory statistics in CantripOS: `allocated_bytes` and `allocated_objects`.

The reasoning behind `allocated_objects` is very simple: the watermarking mechanism in seL4 (Section 2.2.1) means that freeing single objects doesn't always guarantee that more memory can be allocated in a particular untyped slab. Only once all objects have been freed, is the slab reset and does all memory become available again. By keeping track of the allocated object count in each slab (untyped object), the system knows exactly when a previously full slab becomes fully empty. Additionally, this is useful to track because allocation strategies can now be considered that try to place short-lived/similar lifespan objects in the same slab to hopefully allow a slab reset, and thus use memory more efficiently.

`allocated_bytes` stores the total number of allocated bytes in a slab, and is only decreased (set to 0) when the count of allocated objects (tracked by `allocated_objects`) drops to 0. The slab is considered full once `allocated_objects` reaches the value of `free_bytes`.

[6]The referred to capability tutorial is available in [40]

Updating alloc() and free()

alloc() is updated to increment the allocated_objects attribute of the UntypedSlab in which a request was allocated by the number of objects that were allocated, and increment allocated_bytes by the total size of all allocated objects, including memory wasted due to alignment (see Section 2.2.2 for more information on alignment).

The free() function is updated to unwrap isLastReference and untypedSlabIndex from the invocation of seL4_CNode_Delete. If isLastReference is true and untypedSlabIndex is non-null, the slab with a matching cptr to the untypedSlabIndex is updated by decrementing allocated_objects by 1 for each deleted object. If the value reaches 0, it also resets allocated_bytes to 0. The correct UntypedSlab is identified through a linear search of the untypeds SmallVec.

3.3 Our Improved Design for the CantripOS Memory Manager

After further study of the system, while we saw the approach described in Section 3.2 is correct, we determined that it might be slow because it introduces an order-linear dependency on the number of neighbours of an object in the seL4_CNode_Delete invocation. This behaviour is a consequence of how the Mapping Database (MDB) doubly-linked list is constructed when allocating new objects: it inserts an entry for the new object in that list directly after the source untyped from which it originates. It leads to, in the worst case, traversing all existing objects in a slab during object deletion. We present an experiment confirming this in Appendix C.1. An example of such a traversal appears in Figure 3.1. This issue was noticed and raised during our discussions with Sam Leffer [38] and Kent Mcleod [41].

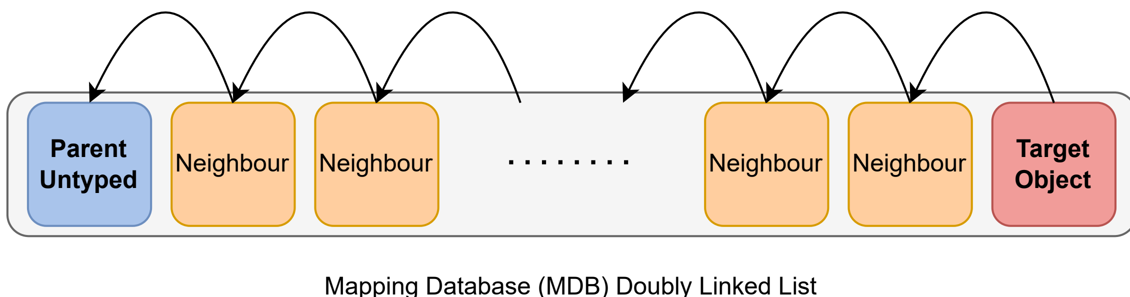


Figure 3.1: An example traversal through the Mapping Database list when identifying the source Untyped Object (parent) of a Target Object. If multiple objects are allocated in that Untyped, the program may need to iterate over an MDB list entry for each object.

After reconsidering the design in Section 3.2, we at a new solution, described below, of O(1) complexity in the user space and without increasing the complexity of the seL4_CNode_Delete invocation in the kernel. It requires more work to take place in the user space - constant in terms of number of allocations and allocated objects, and linear in

terms of number of slabs. In comparison, the seL4 kernel requires very little modification.

The new approach, instead of using the MDB list to identify the parent untyped object, relies on the fact that each object located in an untyped region will have a physical address that is lying between the base and top physical address of the untyped region.

3.3.1 Goals

The goals of this design remain mostly the same as the ones described in Section 3.2.1. The only difference being goal 3, which previously described returning ID of the slab to which memory is returned. The new goal instead is to return the physical address of the object that was deleted (if it was deleted).

3.3.2 What to Return From the Kernel to Improve Memory Bookkeeping?

As in Section 3.2.2, the information needed to maintain proper per-slab granularity bookkeeping of memory remains the same.

3.3.3 Kernel Modifications

Part of the required modification to the kernel overlaps with the one proposed for the previous design, described in Section 3.2.3, but there are a few caveats. The two required modifications are: *returning a single additional value from invocation* and *getting the physical address of the deleted object*.

Returning a single additional value from seL4_CNode_Delete

Instead of returning the index of the untyped object in the top level CNode, the physical address of the deleted object is returned. This value is captured by the parameter `objPaddr`, also of type `seL4_Word`. Upon further inspection of the previous design, it becomes apparent that the boolean `isLastReference` can be "merged into" the other value returned. More specifically, the `objPaddr` represents a physical address, which must be greater than or equal to `PADDR_BASE` and smaller than or equal to `PADDR_TOP` constants[7], both of which are defined per architecture in the kernel. It is therefore safe to reserve a value of `objPaddr` greater than `PADDR_TOP` to represent the `false` value of the original `isLastReference` parameter. The `true` value is implied when `objPaddr` is any valid value. As we are working with a 32-bit RISC-V system, we selected the value of the maximum unsigned 32-bit integer as the "special" value. The Listing 3.4 highlights the portion of the XML displayed in Listing 3.2 that is modified for the approach described here.

```
1 <!-- omitted for brevity --!>
2     <param dir="in" name="depth" type="seL4_Uint8" description="Number of
      bits of index to resolve to find the capability being operated on."/>
3     <!-- Modified content start --!>
```

[7]The constants `PADDR_BASE` and `PADDR_TOP` are defined in the seL4 C kernel implementations in [26].

```
4     <param dir="out" name="objPaddr" type="seL4_Word" description="Physical
       address of the object being deleted. Is set to UINT32_MAX if
      capability deleted is not backed by physical memory, or is not the last
       capability pointing to that object (refcounting)."/>
5     <!-- Modified content end --!>
6 </method>
```

Listing 3.4: Second design: Portion of the modified XML for the `seL4_CNode_Delete` Object Invocation

```
1 exception_t invokeCNodeDelete(cte_t *destSlot, word_t *buffer)
2 {
3     exception_t status;
4     // Returns the objects physical address (for kernel mem: constant
      offset from the virtual memory by PPTR_BASE_OFFSET),
5     // if the capability is pointing to a physical object (backed by
      physical mem), and is the last capability
6     // pointing to this particular object (refcounting). Otherwise, returns
       UINT32_MAX
7     word_t objPaddr = UINT32_MAX;
8     // Only tracked for capabilities which are backed by physical memory
9     if (cap_get_capIsPhysical(destSlot->cap)) {
10        if (isFinalCapability(destSlot)) {
11            objPaddr = (word_t)cap_get_capPtr(destSlot->cap)-
      PPTR_BASE_OFFSET;
12        }
13    }
14    status = cteDelete(destSlot, true);
15    if (status == EXCEPTION_NONE) {
16        setMR(NODE_STATE(ksCurThread), buffer, 0, objPaddr);
17    }
18    return status;
19 }
```

Listing 3.5: Second design: modified `invokeCNodeDelete` function in seL4 kernel.

Getting the physical address of the deleted object

To get the correct value for `objPaddr`, the kernel needs to make sure the capability being deleted is a physical capability, `cap_get_capPtr()` is undefined for other capabilities, and it does not make sense for a non-physical capability to have a physical address (line 9 in Listing 3.5). If it is a physical capability, and it is also the final capability of the pointed-to object, then the physical address of the object is calculated (lines 10 – 11 in Listing 3.5).

The value returned from `cap_get_capPtr()` is a physical pointer – which means it is a pointer to the virtual address space of the kernel. Similarly to the 2.6 version of Linux kernel [11, Section 3.7.1], the seL4 kernel sets up a direct mapping from its virtual kernel address space to the physical kernel address space. This mapping is done by simply subtracting the `PPTR_BASE_OFFSET` from a virtual pointer, and the result is a physical address in the kernel's physical address space. We need to perform this mapping either

here, or later in the user space, as all the pointers to untyped objects handed off at boot time to the `rootserver` (and later, `MemoryManager` thread) are physical addresses of those objects.

If the capability for deletion is not a physical capability, or it is not a final capability, the max untyped 32-bit integer value is returned instead as `objPaddr`. If any issues occur while the capability is being deleted (`cteDelete` function call gets pre-empted, line 14 in Listing 3.5), the message register is not modified.

3.3.4 User space Modifications

In user space, the changes required are very similar to the previous design, described in Section 3.2.4.

The `free()` function, now unwraps the `objPaddr` value from the invocation `seL4_CNode_Delete`. If `objPaddr` is equal to the maximum value of an unsigned 32-bit integer (target system is RISC-V 32-bit), no bookkeeping information is updated. Otherwise, the system iterates over every `UntypedSlab` in the `untypeds` array, and checks if `objPaddr` falls within the physical address boundaries of the untyped object of the slab, by comparing with `_base_paddr` and `_last_paddr` attributes of the slab[8]. When a match is found, the bookkeeping information for that slab is updated: decrementing `allocated_objects` attribute by 1 for each deleted object. If the value reaches 0, `allocated_bytes` attribute is also reset to 0.

3.3.5 Validation of the Designs' Correctness

For both designs implemented, their correctness was validated in terms of impact on the seL4 kernel's correct execution (not a formal proof), and the correctness of the newly added per-slab memory bookkeeping information.

To satisfy the first validation goal, we checked the modified seL4 kernel of the system using the *seL4Test* test suite [42]. The test suite runs small, user level programs checking various features of the kernel via Object Invocations and system calls.

Unfortunately, the user space portion of CantripOS, being a fresh and experimental system, does not have a complete unit test suite. Therefore, to satisfy the second validation goal, we ran extensive stress tests on the CantripOS system, using `shodan_stress.robot` script, which sequentially starts and stops various test applications (some of these applications are introduced in more detail, in Section 4.1.3), along with the later designed synthetic workloads, described in Section 4.1.2. We checked the correctness of the improved memory bookkeeping, by periodically dumping the status of untyped slabs along with the actual (tracked by kernel) current state of each untyped object's watermark[9],

[8] The original implementation of CantripOS did not update the physical address boundaries for untyped objects which were created at boot time from "dirty" untyped objects (e.g. ones containing the system CAmkES components). We uncovered this while implementing this modification, and fixed it accordingly.

[9] The actual, in-kernel, information on each untyped object was obtained using a custom CantripOS `UntypedDescribe` object invocation, which returns exact information on a specific untyped object. It is used for reclaiming untyped objects from the `rootserver` and safely splitting slabs during `MemoryManager` initialisation.

and verifying the watermark position is the same and object count is properly updating.

No implementation of software is immune to the possibility of bugs (apart from the verified configuration of seL4, which has a proof backing this up [23, 24]), and our design was no different. Through the extensive testing, we uncovered and fixed various bugs created along the way, including some which were present in the system beforehand. The latter bugs were more interesting, as they were present in the system before any proposed design changes were introduced. For example, the earlier mentioned bug in incorrect dirty untyped object splitting, described in more detail in Appendix B.1. Additionally, our random synthetic workload (Section 4.1.2) uncovered a small bug in the `sel4sys` Rust: the `size_bits` function returned an incorrect value for `seL4_NotificationObject`[10].

3.4 Implementing the Best-Fit Strategy for Memory Allocation

The currently implemented allocation strategy in CantripOS, *next fit*, is a good choice for the limited information available, as it tries to minimise the amount of failed allocation attempts. However, an implementation of *best fit* strategy would cut down the amount of failed `seL4_Untyped_Retype` Object Invocations down to 0, while potentially decreasing memory fragmentation. Previous research has shown that best fit is a very good allocation policy, but it is often unfairly dismissed as being impossible to implement efficiently [9]. However, since CantripOS uses a fixed number of untyped objects, determined at system initialisation, and seL4's watermarking constraint means that there is always a fixed amount of free blocks that can be considered. This makes best fit a much more attractive allocation policy, with the overhead of each allocation potentially much closer to other sequential fit allocation policies. We measure and report this overhead, as described in Section 4.1.1.

The actual implementation of the best fit allocator is very simple, on a high level it involves the following:

1. Iterate over each object descriptor in an object descriptor bundle.

2. For each untyped slab, compute the space that is left usable on the slab after allocation. The computation includes alignment constraint fragmentation, but does not try to minimise it; it only minimises the space left above the watermark, which is the leftover memory.

3. Attempt to allocate on the slab, which would lead to minimal "leftover" memory in a slab. This operation can only fail if an out of memory error were to occur, or the bookkeeping of memory is incorrect.

4. On successful allocation, update the bookkeeping information for the untyped slab.

[10]This tiny change has already made it upstream to the Open Se Cura repo.

3.5 Infrastructure for Performance Analysis

In order to assess the system's performance, we propose and implement several workloads with varying patterns of memory allocation and deallocation requests. The exact details of workloads used are presented in Sections 4.1.2 & 4.1.3. Creating, running and collecting data from evaluations using these workloads requires additional software infrastructure within the CantripOS system, outlined below. In the interest of brevity, this outlined will be limited to high-level description of modification.

The `MemoryManager` along with `MemoryManagerStats` structs are amended to keep track of additional metrics, such as: watermarking constraint fragmentation per slab, alignment constraint fragmentation, total occupied memory per slab.

We extend the IPC interface for `MemoryManager` to allow for runtime selection of allocator mechanism, enabling/disabling tracing and enabling/disabling logging of retired instruction count per `alloc()` and `free()` function call.

Further, we extend the `DebugConsole` to include user-accessible commands for running and configuring synthetic workloads, enabling memory allocation request tracing and dumping these traces to console, replaying these traces and also running pre-made representative workloads using traces of real applications, as described in Section 4.1.3.

Chapter 4

Performance Evaluation

4.1 Experiments

We consider two kinds of workloads for performance analysis of CantripOS's memory management system, synthetic (Section 4.1.2) and representative (Section 4.1.3), each motivated by different goals.

G1. Range of possible behaviours. The first goal is to explore the range of possible behaviours of the implemented memory allocators. These types of workloads are not representative of real programs, and produce varied request patterns. This is not a coincidence or an oversight, but deliberate by design, as such synthetic traces have been broadly used for measuring relative performance of different dynamic memory management algorithms [43]. Analysing the memory fragmentation outcome of these workloads is expected to give further insights into behaviour of next fit and best fit allocation strategies in the specific, restricted environment of CantripOS and seL4.

G2. Behaviour under real workloads. The second goal, is to analyse the memory allocators' behaviour when subject to workloads representative of real applications. As CantripOS, at the time of writing this, is a new and still experimental system, there are no such applications available yet. Instead, there are some sample applications, which do real work (spawning a thread with a heap, using timers, I/O and loading and calling Machine Learning models) but not yet useful work. They are however the closest thing to an actual target workload for this system, the exact method of how they are used for evaluating the performance of CantripOS's memory management system is detailed in Section 4.1.3. Additionally, Wilson et al. [5], describe common patterns of memory requests observed in program behaviour. This work attempts to reconstruct the behaviours described to further explore memory allocator behaviour in a realistic-like setting.

G3. Measure latency of allocation and deallocation. The third goal, is to identify what overhead in terms of executed instructions, if any, does the introduction of best fit algorithm have. An additional subgoal is to identify whether the number of failed retyping object invocations has an impact on the performance of the original, next fit allocator.

4.1.1 Measured Metrics

For all experiments, we collect data on the following metrics:

Alignment constraint fragmentation is the amount of memory lost due to the alignment constraint, as described in Section 2.2.2. This metric is collected at a per-slab granularity.

Watermarking constraint fragmentation is the amount of memory lost due to the watermarking constraint, as described in Section 2.2.1. This metric is collected at a per-slab granularity.

Retired CPU instruction count is the number of CPU instructions that were executed by the CPU up to the point of measurement. For RISC-V cores, this value is constantly updated in the `instret` CSR register and can be obtained using the `RDINSTRET` pseudo-instruction [44, Section 2.8]. This metric is used to determine latency of an `alloc()` and `free()` function calls in the user space, as per the goal **G3**.

Failed retype invocation count is the number of failed `seL4_Untyped_Retype` object invocations.

Out of Memory (OOM) count is the number of OOM errors that were thrown during a workload's execution.

Number of slab resets tracks the number of times an untyped object's watermark was reset to 0. These resets represent reuse of previously allocated and freed memory, and should indicate potentially improved efficiency of memory utilisation under seL4's watermarked physical memory allocator.

4.1.2 Synthetic Workloads

Most early research papers looking into workloads used for measuring and evaluating memory allocator performance relied on using completely random object sizes and lifetimes. This was partially based on false assumptions that size and lifetime distributions of memory allocations are independent [5]. Both Zorn and Grunwald [43] as well as Johnstone and Wilson [9] have shown that the regularities in size and lifetime distributions of memory requests play an important role in the fragmentation behaviour of allocators.

However, although synthetic workloads cannot easily replicate such distributions of real applications, they are a useful tool for meeting the goal of *exploring the range of possible behaviour of allocators* (**G1**). These workloads also have much more allocation and deallocation requests than representative workloads, so they allow for more accurate *measurement of latency of allocations and deallocations* (**G3**).

Randomly distributed workloads

The flow of this workload is presented in Figure 4.1. In this implementation, such a workload is characterised by: a *random integer seed*, *total number of memory requests* and *percentage chance for deallocation* (called: *free chance* in Figure 4.1). The *random integer seed* is used for initialising the pseudo-random number generators for reproducibility, *total number of memory requests* is how many allocation and deallocations combined will the

workload attempt and *percentage chance for deallocation* determines the likelihood of a random object being deallocated while the workload is running.

At each step of the workload, a random value from 1-100 is drawn, and if it's smaller than the *percentage chance for deallocation*, a random object to be freed. Because we are working in a memory-constraint environment of embedded systems, there is also a hard upper limit of 1000 objects live at the same time. If this amount is ever reached during a random trace simulation, regardless of the random value drawn, an object descriptor is deallocated. This is to avoid system failure due to running out of heap memory. If an allocation is to be performed instead, a random object type out of 9 possible physical seL4 objects is chosen: `seL4_UntypedObject`, `seL4_CapTableObject`, `seL4_SchedContextObject`, `seL4_SmallPageObject`[1], `seL4_TCBObject`, `seL4_EndpointObject`, `seL4_NotificationObject` or `seL4_PageTableObject`. For the first four types, additionally a random size for that object is chosen – for the page object rather than size, this value is the number of consecutive pages. After the number of requests matches the *total number of memory requests* (*workflow len* on the diagram), the workflow makes free requests for all remaining objects and exits.

There are a total of 4 experiments performed for this study, using the randomly distributed workloads, repeated 3 times to collect all the data described in Section 4.1.1, for both the original next fit implementation and the best fit implementation described in Section 3.4. The parameters for the experiments are summarised in Table 4.1. As further highlighted in Section 4.2.1, the parameter capturing *percentage chance for deallocation* was originally exponentially spaced in powers of 2, but 48% was later added (mid-point between 32 and 64) for a more holistic view of the system behaviour.

seed	total request count	perc. chance for deallocation	allocator
42	1000	{16, 32, 48, 64}%	{best fit, original next fit}

Table 4.1: Synthetic traces: Randomly distributed workload experiment parameters.

4.1.3 Representative Workloads

In order to study behaviour of the newly implemented allocator in real world scenarios, we utilise the existing limited number of applications performing real work. Experiments using these workloads aim to meet the goal of analysing the allocators' *behaviour under real workloads* (**G2**). Four applications were chosen: `hello`, `timer`, `fibonacci` and `mltest`. Each application in CantripOS always runs on a separate thread. The exact allocation and free requests performed by each app were traced during the runtime of those applications by the tracing mechanism described in Section 3.5, inspired by earlier papers on dynamic memory allocation evaluation [5, 9]. These traces allow for easy reproducibility, much faster replication of the exact behaviour of specific applications and creating hybrid traces for evaluating future features of CantripOS. Each application is an example of a ramp

[1] This is an architecture-agnostic abstraction, for RISC-V 32-bit the exact object name is: `seL4_RISCV_4K_Page`.

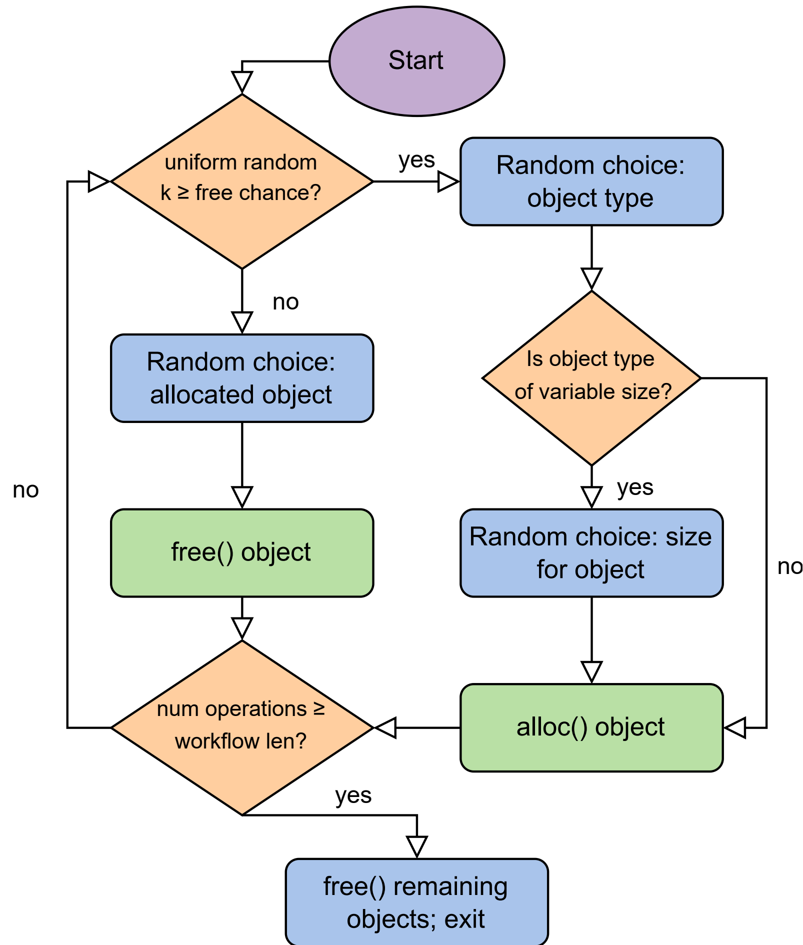


Figure 4.1: Simplified diagram of the uniform synthetic random workflow. Omitted in the diagram: on first step, if there are no allocated objects – always allocate, if allocated object count exceeds 1000 – always deallocate.

behaviour app [5], allocating memory at the start of its execution and only deallocating on exit, except for `mltest`, which deallocates a large chunk of memory during its execution, but does not allocate any memory afterwards hence it is not visible in the app's memory profile (Figure 4.5).

Standalone application trace workloads

These workloads replay each application's individual trace, measuring memory fragmentation. As the applications are not very complex by themselves, we expect these workloads to show minimal fragmentation across both allocators.

The `hello` application is a minimal C app, which simply prints two messages to the debug console, after which it loops forever calling `seL4_Yield` system call on each iteration, donating the rest of its time slice to a different thread. Source code of the application is presented in the Listing 4.1. The app's peak memory footprint is exactly 46,400 bytes across 15 different objects. Memory profile of this application is presented in Figure 4.2.

```c
#include <cantrip.h>
```

```
int main() {
    debug_printf("\nI am a C app!\n");
    debug_printf("Done, sleeping in infinite loop\n");
    while (1) {
        seL4_Yield();
    }
}
```

Listing 4.1: `hello` C app source code.

The `timer` application is a slightly more complex Rust app, it uses the CantripOS's runtime SDK and starts 2 different timers. After the timers expire, the application exits. Source code of the application is available on the Open Se Cura repo. The app's peak memory footprint is exactly 104,448 bytes across 29 different objects. Memory profile of this application is presented in Figure 4.3.

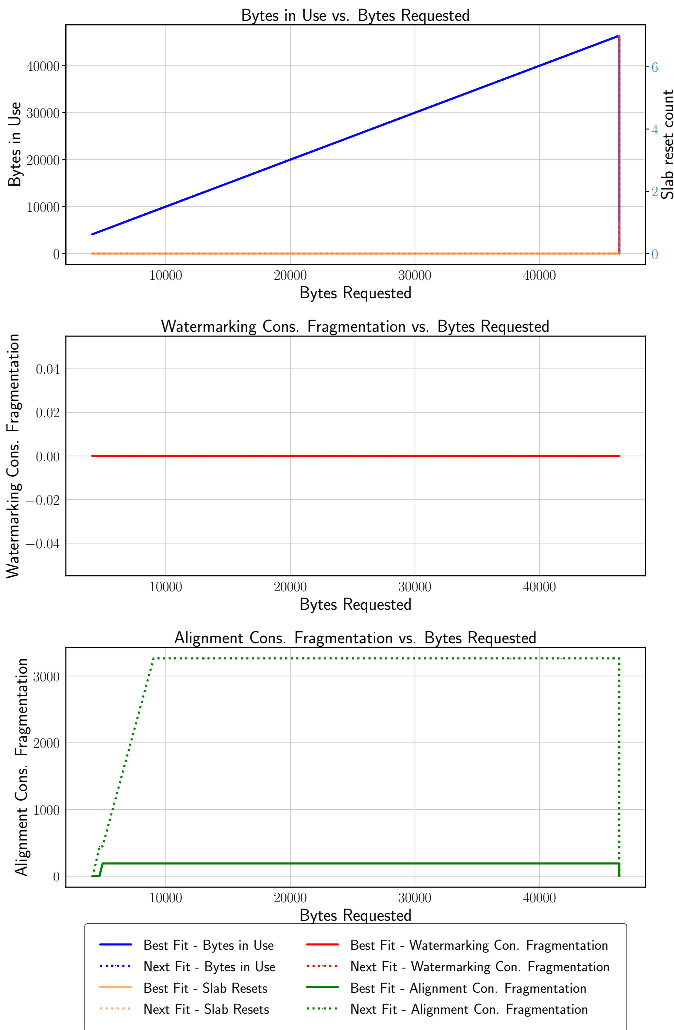


Figure 4.2: `hello` C app: memory profile.

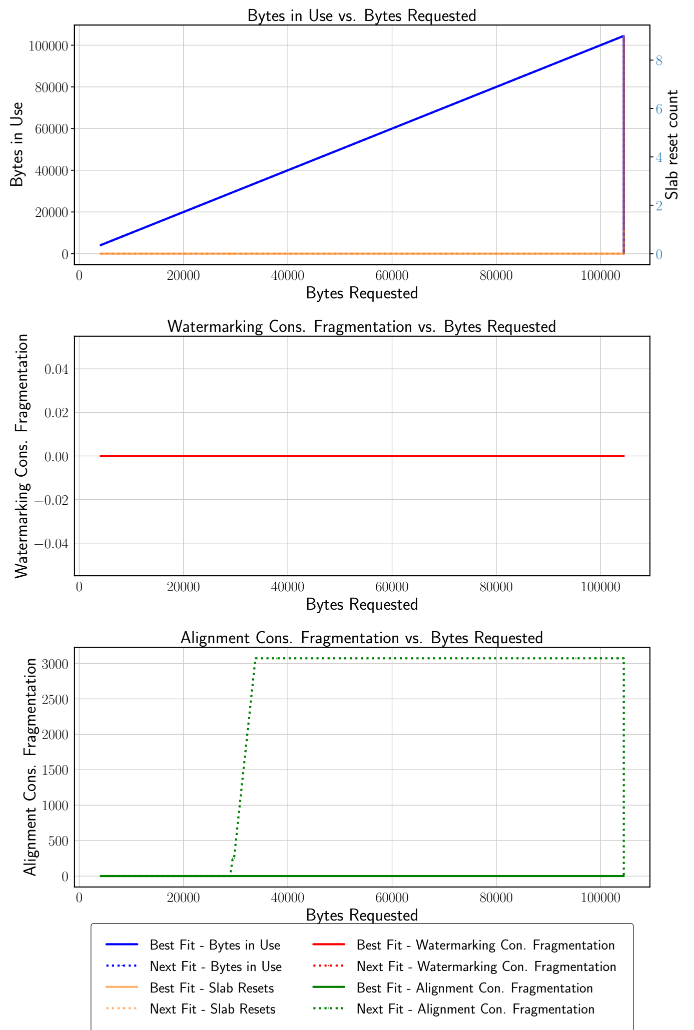


Figure 4.3: `timer` Rust app: memory profile.

The `fibonacci` application is another C app, which prints the first 80 Fibonacci numbers to the console, waiting for 200 interrupts between each number. The program loops

forever, until it's killed. Source code of the application is available on the Open Se Cura repo. The app's peak memory footprint is exactly 208, 896 bytes across 58 different objects. Memory profile of this application is presented in Figure 4.4.

The `mltest` application is a Rust app, and the only one which uses the system's ML core (Kelvin). It is also the only app which during its runtime performs both memory allocation and deallocation requests. It starts by loading an example model into memory (roughly 1 MiB in size), offloading it to the ML core and freeing the objects required for storing the model, then prints out the model's output 10 times in 1s intervals. Source code of the application is available on the Open Se Cura repo. The app's peak memory footprint is exactly 1, 157, 376 bytes across 285 different objects. Memory profile of this application is presented in Figure 4.5.

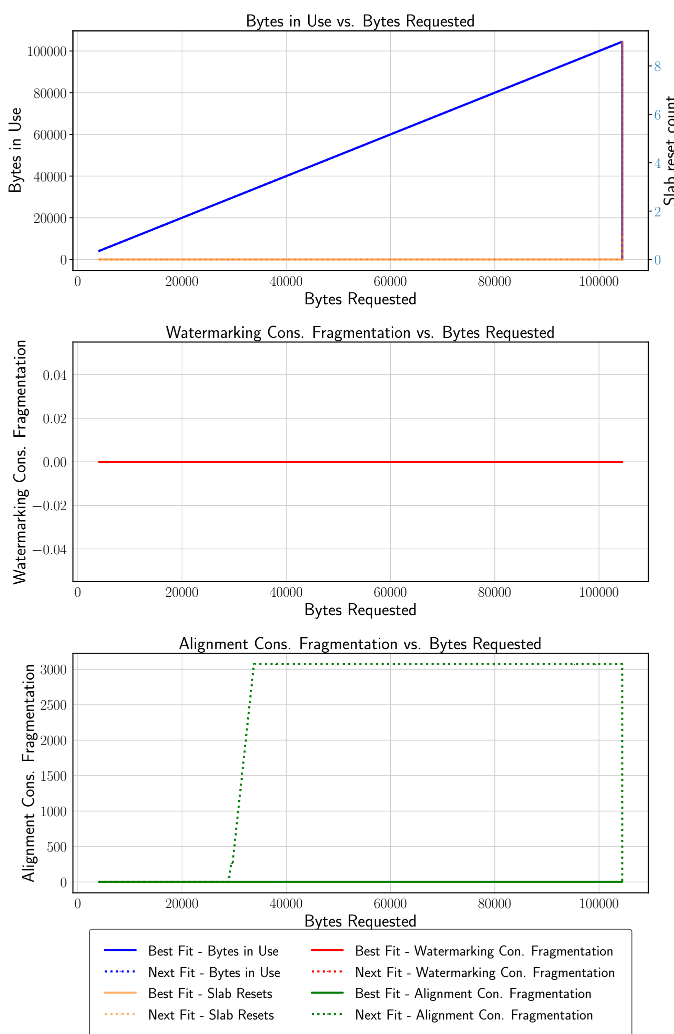


Figure 4.4: `fibonacci` C app: memory profile.

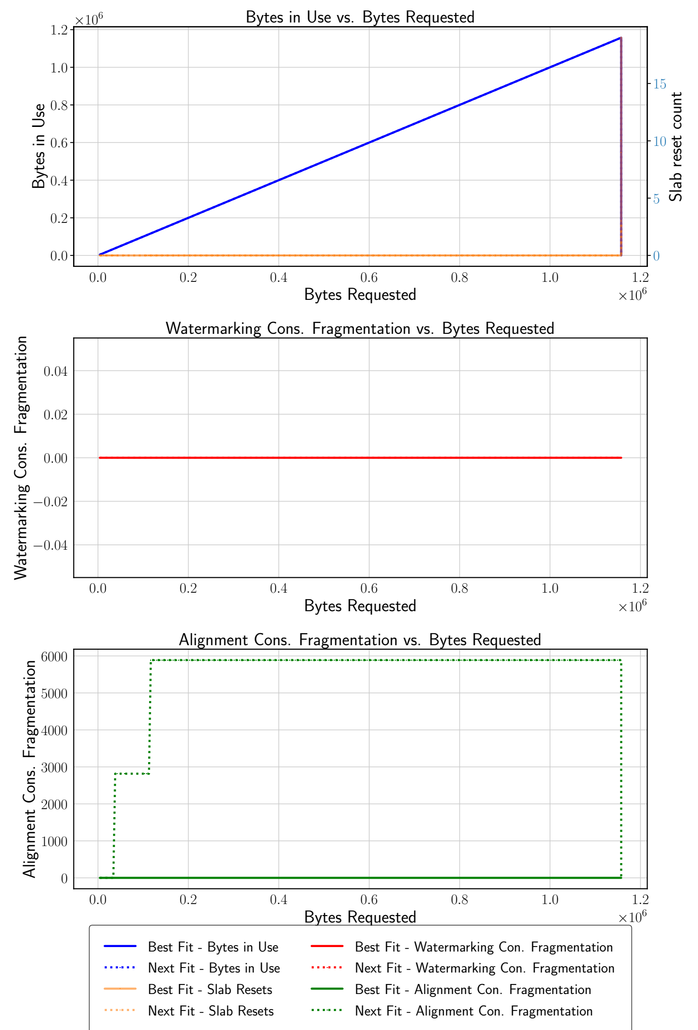


Figure 4.5: `mltest` Rust app: memory profile. This app also uses the ML core.

Sequentially interleaved application workloads

In the current state of CantripOS, the applications can be started and stopped sequentially from the CLI, using the `start <app_name>` and `stop <app_name>` commands. Each application has therefore two distinct events: starting and stopping it. A sample, realistic workload can therefore be launching and stopping some applications in different orders. Since we used 4 applications for the performance evaluation, there are 8 total events that can occur. However, early experimentation has shown that interleavings with only 8 events lead to very limited results and extremely easy workloads for both allocators, resulting in almost no fragmentation, hence we decided to use 8 applications instead. The additional four are copies of the **hello**, **timer**, **fibonacci** and **mltest** apps, leading to a total of 16 start/stop events. Since the space of possible interleavings has factorial growth (for 16 events, there are over 81 billion possible interleavings), exploring all possible arrangements is intractable, hence we settled on exploring the behaviour of 3. All 3 sequences are presented in Table 4.2.

sequence name	sequence
1. Multiple peaks behaviour	$hello\ hello_s\ timer\ timer_s\ mltest\ mltest_s\ fibonacci\ fibonacci_s\ hello2\ hello2_s\ timer2\ timer2_s\ mltest2\ mltest2_s\ fibonacci2\ fibonacci2_s$
2. Large ramp behaviour	$hello\ timer\ mltest\ fibonacci\ hello2\ timer2\ mltest2\ fibonacci2\ fibonacci2_s\ mltest2_s\ timer2_s\ hello2_s\ fibonacci_s\ mltest_s\ timer_s\ hello_s$
3. Select random interleaving	$hello\ timer\ mltest\ timer_s\ mltest_s\ fibonacci\ hello2\ timer2\ hello_s\ fibonacci_s\ mltest2\ fibonacci2\ fibonacci2_s\ mltest2_s\ hello2_s\ timer2_s$

Table 4.2: Representative traces: Sequentially interleaved application workloads. app_name_s indicates this is the exit (stop) event of the application. app_name2 indicates a copy of *app_name* trace is used.

Sequence 1: Many peaks behaviour [5], the trace simulates each application being started then stopped one after the other, resulting in 8 individual "peaks" of memory usage. This workload is expected to produce results exactly the same as the Figures of each standalone application, concatenated in the order described in Table 4.2 and repeated twice. The memory profile of this sequence is presented in Figure 4.12.

Sequence 2: Large ramp behaviour [5], the trace simulates each application being started, then after all applications are "running" – memory for them is allocated, each application is stopped in the reverse order. This workload is expected to produce more alignment constraint fragmentation than *Sequence 1*, as more memory is used, so there is a higher chance for different-sized allocations to land in the same slabs. It is also expected to produce some watermarking constraint fragmentation, as `mltest` app frees memory during its runtime, and it is possible that the slab from which these allocations are freed also had some other objects present. The memory profile of this sequence is presented in Figure 4.13.

Sequence 3: Select random interleaving, the trace simulates a random interleaving,

without replicating a particular pattern. It produces memory requests equivalent to starting 3 applications, stopping 2, starting 3 more, stopping another 2, starting the final 2 application and in the end stopping the remaining 4. This workload is expected to produce both alignment constraint fragmentation and watermarking constraint fragmentation, since allocations and deallocations of apps are mixed. The memory profile of this sequence is presented in Figure 4.14.

4.1.4 Expected Study Outcomes

Based on previous literature on memory allocators and our experience with the CantripOS system, we expect the following hypotheses to hold for the results of synthetic workloads as well as representative workloads:

H1. Best fit allocator produces smaller peak and average fragmentation than next fit. This hypothesis is based on the idea that next fit starts by placing all objects in a single large slab, hence alignment constrain fragmentation can be introduced as different sized objects will mix on the same slabs, and larger watermarking fragmentation can be introduced as allocated objects have random lifetimes. Best fit on the other hand should try to "pack" smaller objects into smaller slabs and pack objects of the same size together, as they are more likely to produce perfect fits. It is further motivated by evidence from previous studies on allocators using both synthetic and representative [9] workloads is systems without the watermarking constraint imposed by seL4, in which best fit outperformed next fit.

H2. Untyped slabs fill up from smallest to largest for best fit, but in reverse order for next fit. This hypothesis is based on the fact that best fit should prefer memory blocks that fit an allocation request as closely as possible. Since untyped slabs vary a lot in sizes, the smallest slabs will be picked first, as even big objects lead to a lot of left over memory for large slabs, and little for small slabs. Next fit, on the other hand, tries to allocate objects on the last successful slab. Since CantripOS sorts untyped slabs in descending order of sizes, large slabs will receive all allocations first.

H3. Peak instruction count-driven latency of best fit allocator is smaller than next fit. This hypothesis is based on the idea that the original implementation of next fit allocator in CantripOS has "spiky" behaviour, due to some allocation requests causing multiple failed object invocations (Section 3.1.2). Best fit should present a much flatter instruction count distribution, possibly with small deviations when pre-empted or an out of memory occurs.

H4. Average instruction count-driven latency of best fit allocator is smaller than next fit. Similarly to **H3**, this hypothesis results from the "spiky" behaviour of original next fit. Although when next fit succeeds on the first try, execution of best fit allocation might take longer, the frequency of those failed allocations should lead to a higher average latency for next fit.

H5. Use of best fit allocator results in higher number of untyped slab resets. This hypothesis is based on **H2** – since smaller slabs are filled up first by best fit,

there are fewer objects that need to be freed on that slab to reset it. Therefore, there is a higher chance it will be reset than a larger slab used by next fit.

H6. Higher number of untyped slab resets results in lower overall fragmentation. This hypothesis is based on **H5** and assumes that when an untyped slab is reset and memory is freed to the system, total fragmentation decreases. Hence, if a lot of slabs are being reset, the fragmentation should be only spiking slightly and quickly dropping.

4.2 Results

In this section, we discuss the results of performing the experiments described in Section 4.1. Since in the domain of memory allocation, physical time of allocations has little meaning[2], inspired by Wilson et al. [5], the plots presented in this section instead use the total number of bytes requested as the "time" of the allocation. Because the added infrastructure for saving memory traces and later replaying them requires for the system to use slightly more memory than the original binary, hence the size was fixed for all experiments at 1291 KiB. The total available space for allocation in all performed experiments is $2,723,584$ B – approximately 2.6 MiB.

4.2.1 Synthetic Workloads – Random with Uniform Distribution

Initially, out of the 4 experiments described in Section 4.1.2, only 3 were performed with exponentially increasing percentage chance for deallocation, in powers of 2: 16%, 32% and 64%. Figures 4.6, 4.7 and 4.9 show memory profiles of these traces, along with watermarking and alignment constraint, untyped slab resets and highlights when out of memory occurred. For both 16% and 32% (Figures 4.6, 4.7), the small percentage chance for deallocation lead to the system's memory filling up and causing out of memory errors. Interestingly, out of memory occurs much faster for next fit allocator than best fit (around 2.05 and 2.06 MiB of requested memory for next fit, 2.27 and 2.50 MiB for best fit), suggesting better memory utilisation using best fit allocator. On the other hand, 64% workload's memory profile (Figure 4.9) is very spiky, both in terms of used bytes and fragmentation, often dropping to zero. Such cliff behaviour between the percentages 32 and 64 led us to analyse a fourth trace, for 48% chance of deallocation (Figure 4.8). Both the 48 and 64 percentage chance traces reach much smaller peak memory usage (approx. 0.19 MiB and 0.06 MiB) in comparison to the other two runs (approx. 1.72 MiB for 16% run and 1.44 MiB for 32% run) and never reach an out of memory scenario.

Some hypotheses stated in Section 4.1.4 are supported by these results, the hypothesis **H1**, states that best fit allocator produces smaller peak and average fragmentation than next fit, which is confirmed just by visual inspection of these graphs: for all 4 runs both watermarking and alignment constraint fragmentation is almost always lower for best fit. Further, Table 4.3 shows peak and average fragmentation results for the four different

[2] Reasoning behind this is simple: it is not the physical time of memory requests that impacts fragmentation but their **ordering**.

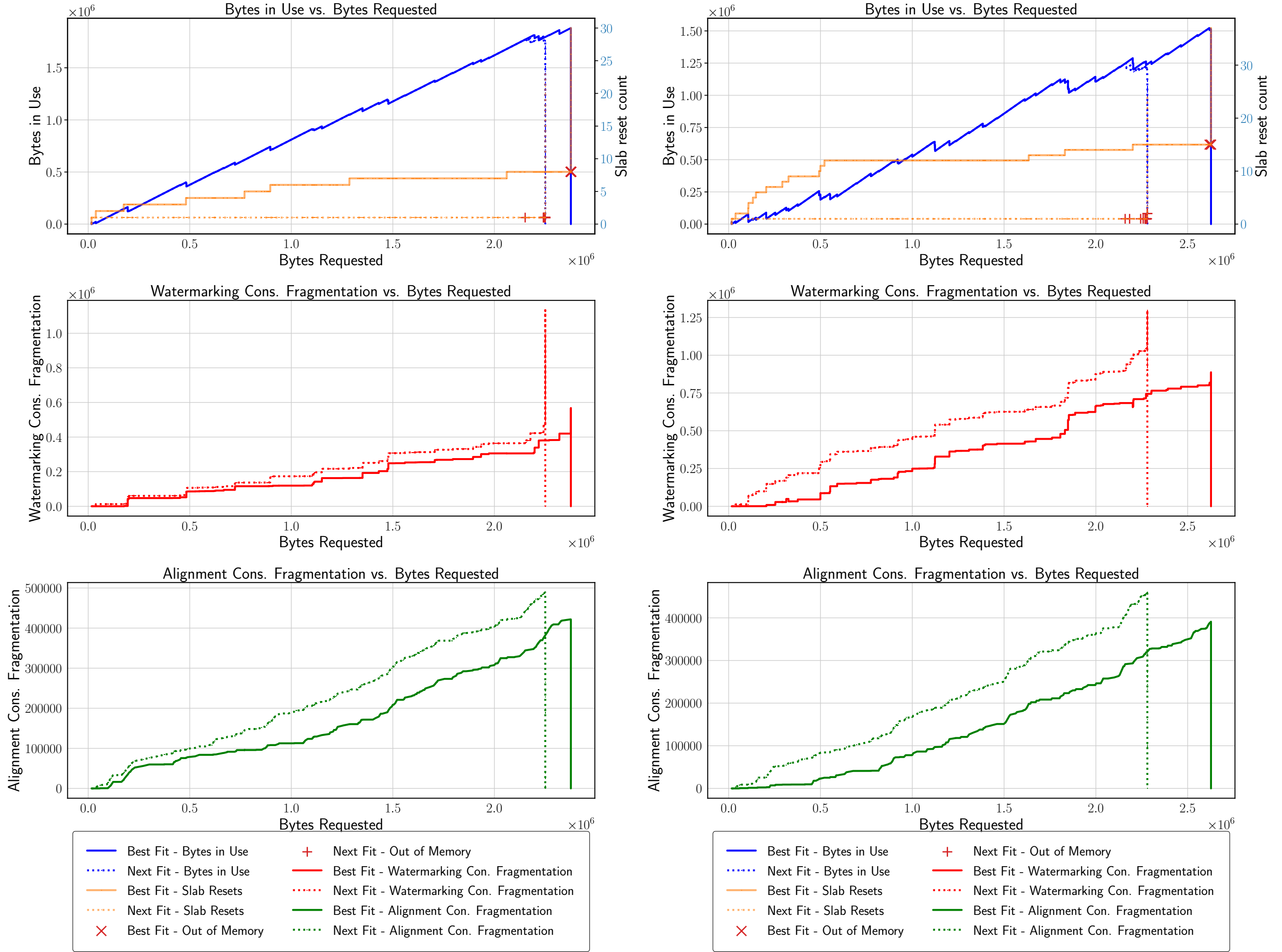


Figure 4.6: Memory profile for a random uniform run. Seed 42, no. operations: 1000, free chance: 16%.

Figure 4.7: Memory profile for a random uniform run. Seed 42, no. operations: 1000, free chance: 32%.

runs, and for all 4 the total peak and average fragmentation is between 1.4 to 17 times smaller. Additionally, alignment and watermarking constraint fragmentation is always smaller for best fit when compared to next fit.

Such performance is expected based on the evident confirmation of hypothesis **H5**: use of best fit allocator results in higher number of untyped slab resets, and the resulting hypothesis **H6**: higher number of untyped slab resets results in lower overall fragmentation. Figures 4.6 – 4.9 show how the number of untyped slab resets (orange lines) is much higher for all 4 traces. Table 4.4 presents the numerical values, which also confirm this.

Figures 4.10 and 4.11 display snapshots of untyped memory slabs at 25th, 50th, 75th and 100th percentiles of synthetic random traces with 16 and 48 percentage chance for deallocation, for both best fit allocator (solid filled in bars) and next fit (hatched bars). The untyped slabs are sorted in descending order by size. As the traces progress, it is clearly visible that using the best fit allocator spreads its allocations among the smallest slabs first, before "spoiling" the larger slabs, a heuristic observed by Wilson et al. [5],

Metric	**16% chance**		**32% chance**		**48% chance**		**64% chance**	
Allocator:	best fit	next fit	best fit	next fit	best fit	next fit	best fit	next fit
Peak fragmentation (in KiB)								
watermarking:	553.80	1106.69	**865.92**	**1267.75**	282.95	785.69	36.34	72.16
alignment:	**411.77**	**477.67**	381.89	449.03	67.78	193.91	0.67	15.98
total:	951.66	1584.36	**1237.80**	**1691.00**	340.09	979.59	36.44	81.97
Average fragmentation (in KiB)								
watermarking:	281.05	407.09	**442.09**	**726.41**	79.02	306.84	0.42	6.03
alignment:	**231.73**	**299.89**	184.08	267.21	11.03	80.35	0.02	1.76
total:	512.79	706.98	**626.17**	**993.61**	90.05	387.19	0.45	7.79

Table 4.3: Peak and average memory fragmentation for synthetic randomly distributed traces, with different percentage chances for deallocation. Total is computed as the peak/average of the sum of both fragmentation types (not the sum of the peak/average values). Highest value in each category is **bolded** for best and next fit.

mentioned in Section 2.1.1 on sequential fits. Next fit, due to a descending ordering of slabs by size, starts by allocating objects on the bigger slabs and, as watermarking fragmentation and occupied space on the slabs fills up, "spills over" to the smaller slabs. This supports the hypothesis **H2**, stating that untyped slabs fill smallest to largest for best fit, and largest to smallest for next fit.

Such behaviour could also be one of the reasons for best fit allocator producing lower fragmentation for synthetic traces. Deallocating an object on a smaller slab has a higher chance of resetting the slab it is on, as slabs are only reset when all objects in a slab are removed and smaller slabs can fit fewer objects. The increased number of resets in turn mean any unusable memory due to fragmentation is returned to the untyped objects and once again available for use by the system.

Metric	**16% chance**		**32% chance**		**48% chance**		**64% chance**	
Allocator:	best fit	next fit	best fit	next fit	best fit	next fit	best fit	next fit
Number of slab resets								
total:	30	23	37	24	158	13	422	238
per 100 allocs:	3.55	2.71	5.17	3.35	29.81	2.45	84.06	47.41

Table 4.4: Number of slab resets and for synthetic randomly distributed traces. Presents both the total number and average number of slab resets per 100 allocation requests for both best and next fit runs.

4.2.2 Representative Workloads – Standalone Application Traces

Traces of `hello`, `timer` and `fibonacci` apps presented the same allocation patterns, albeit with different sized objects, types and counts, always allocating memory at initialisation. The only app which deallocates memory before exiting is `mltest`, as described in Section 4.1.3. We expected these allocation patterns when originally inspecting all the currently available CantripOS applications. Further, we believed that each application would have no watermarking constraint fragmentation, and this is indeed visible in Figures 4.2

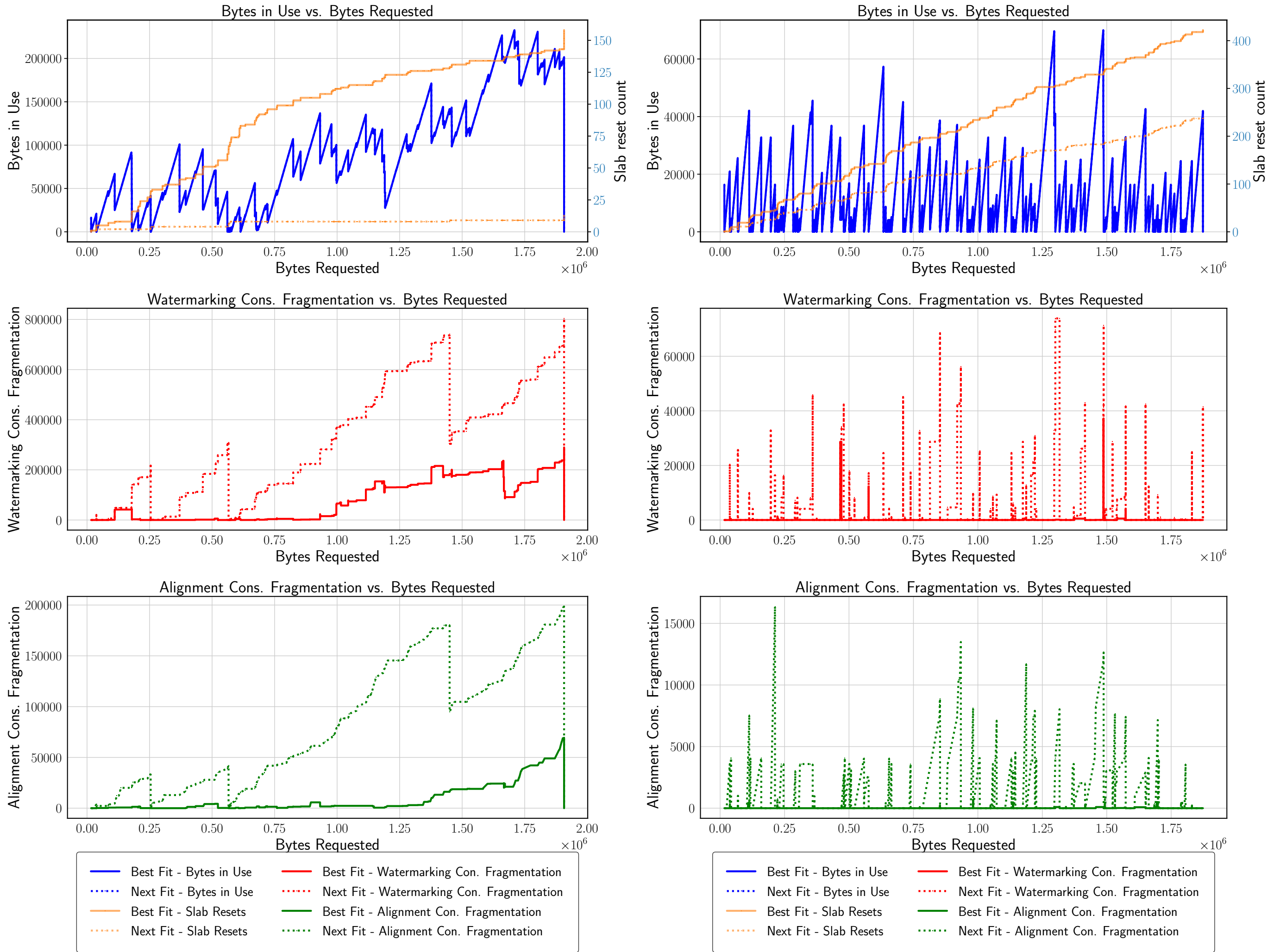


Figure 4.8: Memory profile for a random uniform run. Seed 42, no. operations: 1000, free chance: 48%.

Figure 4.9: Memory profile for a random uniform run. Seed 42, no. operations: 1000, free chance: 64%.

– 4.5.

We also suspected that although next fit should produce some alignment constraint fragmentation, best fit's policy would manage to pack all objects of different sizes into different untyped slabs, resulting in no such fragmentation. The explanation for application traces allocated with next fit producing some alignment constraint fragmentation is simple: Since all applications in CantripOS run in separate seL4 threads, they require as a bare minimum: a CNode to hold its capabilities in, a Thread Control Block object, Scheduling context object, Page table objects to map physical pages to virtual and finally physical page objects to hold user space memory. Each of these objects have different sizes, so when they are allocated contiguously on the same slab, the seL4 microkernel needs to adjust their addresses of allocation to be object-size aligned, fragmenting the memory for each mismatch.

To our surprise, the assumption for best fit producing no alignment constraint-caused fragmentation held for all but the simplest application: `hello`. This app created 192 bytes

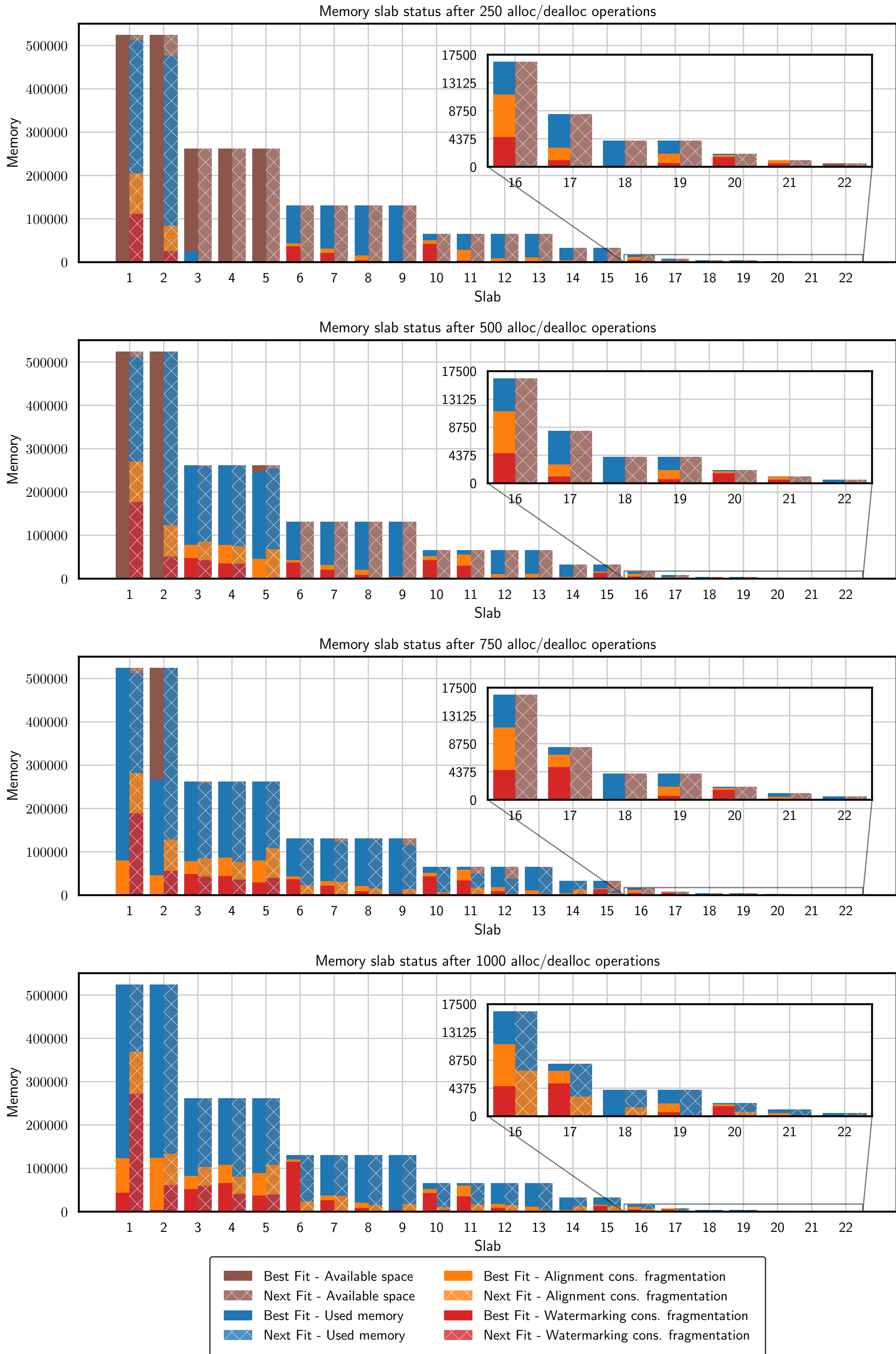


Figure 4.10: Per-slab statistics for a random uniform run. Seed: 42, no. operations: 1000, free chance: 16%.

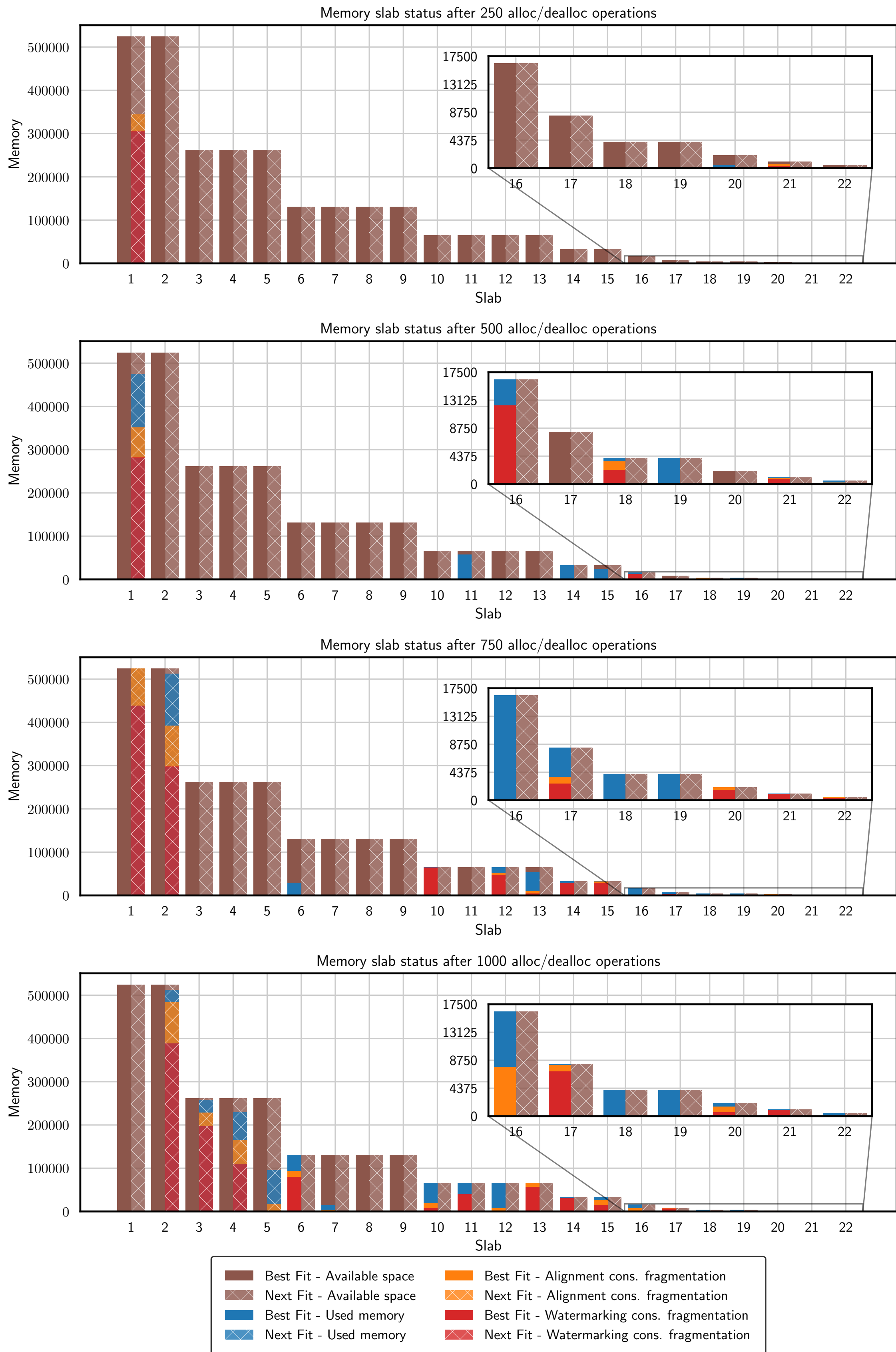


Figure 4.11: Per-slab statistics for a random uniform run. Seed: 42, no. operations: 1000, free chance: 48%.

of alignment constraint fragmentation towards the beginning of its allocation. The trace for `hello` starts by allocating a single physical page, which is placed on a single 4096 byte slab. The reader might find it helpful to look at the system slabs on Figure 4.10 or 4.11 – that would be slab 19. Afterwards, a CNode object (`seL4_CapTableObject`) with 4 slots is allocated, 64 bytes in size. This object is placed in the smallest possible slab of 512 bytes (slab 22), leaving 448 bytes free on the slab. The next allocation is a TCB object (`seL4_TCBObject`) of size 256 bytes for RISC-V 32-bit architecture. This allocation is placed on the next smallest slab, of size 1024 bytes (slab 21), leaving 768 bytes free. Finally, a schedule context object (`seL4_SchedContextObject`), of size 256 bytes, is requested. If placed on slab 21, it wouldn't need to be aligned as the previously placed TCB object is also of size 256, and the resulting "left-over" memory would be $768 - 256 = 512$ bytes. However, if the object is placed on slab 22 instead, since there is a 64 byte object present on the slab, the watermark of the slab is adjusted to 256 bytes for alignment, and there are 0 bytes of "left-over" memory. This is smaller than the 512 bytes left over on slab 21, hence best fit allocator places the object here. The alignment constraint fragmentation for this workflow comes exactly from this allocation: adjusting the watermark requires wasting $256 - 64 = 192$ bytes of memory.

Although the above example shows that even for simple workloads, best fit mechanism can still produce some fragmentation, it is much better when compared to next fit. However, it seems that taking into account memory wasted due to alignment when determining the best slab could be investigated (although even for this simple case, if memory lost due to alignment would be counted towards "left-over" memory for best fit policy, the exact same allocation pattern would occur).

Concluding, the results for standalone application traces, support the hypothesis of best fit allocator producing smaller peak and average fragmentation than next fit (**H1**).

4.2.3 Representative Workloads – Sequential Application Trace Interleaving

These set of traces were designed in hope of achieving more challenging allocation request sequences while remaining in the domain of representative workloads. The 3 sequences introduced in Section 4.1.3 are visible in Figures 4.12, 4.13 and 4.14.

The first trace is a sequence of 4 applications (`hello`, `timer`, `mltest` and `fibonacci`) being started then stopped one after another, repeated twice. This workload forms multiple*peaks* of memory usage, but do not show the same fragmentation issues as described in earlier work [5], as after each peak memory is completely deallocated. The expected result is for the system to behave exactly the same as if the memory profile graphs (both memory usage and fragmentation) in Figures 4.2 – 4.4 were connected together, in a single, larger graph. This exact series of events is presented in Figure 4.12 (except for slab resets accumulating over time). Although trivial, such behaviour is very much expected of a deployed CantripOS system: a compute job is requested from the system (e.g. via an interrupt on an external sensor), memory is allocated for an application to handle the

interrupt, an application runs and exits, deallocating all of its memory, system waits for the next request. With such insight into the behaviour of the system, it is simple to notice that the only possibility for this trace to differ from multiple standalone application traces “chained” together is if the memory bookkeeping implemented in Section 3.3.4 or the implementation of the best fit allocator (Section 3.4) is incorrect.

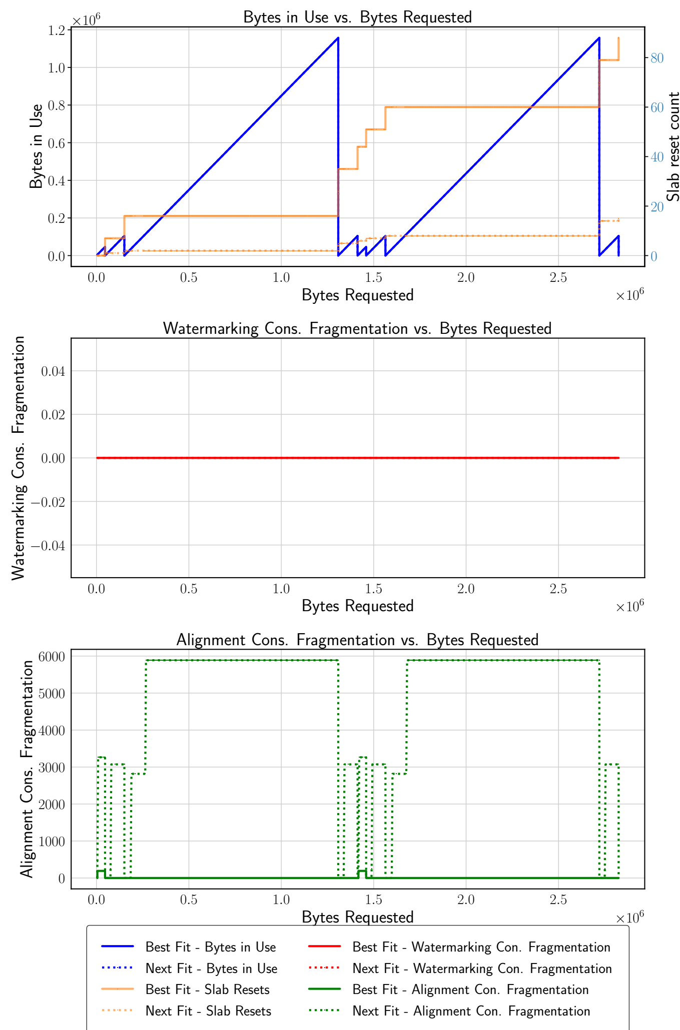


Figure 4.12: Memory profile of sequential app interleaving. Sequence 1: Each app started and stopped one after the other.

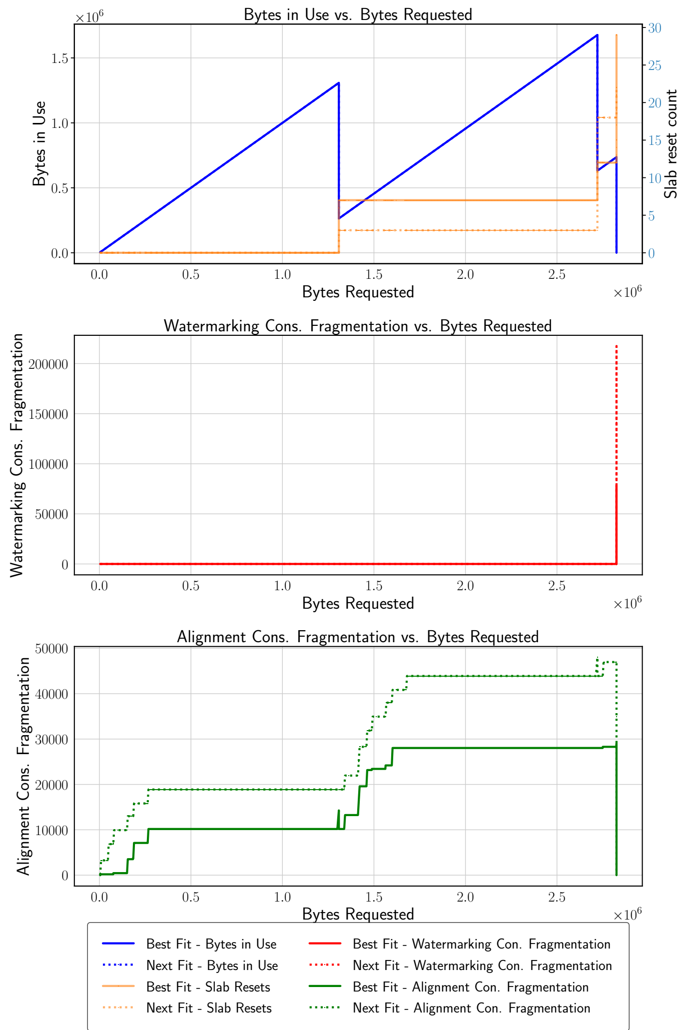


Figure 4.13: Memory profile of sequential app interleaving. Sequence 2: Ramp behaviour, each app started, then all app stopped.

The second trace, referred to as a “large ramp” in Section 4.1.3, allocates (starts) applications in the same order as sequence 1, but deallocates (stops) at the end of all allocations. It is characterised by 3 spikes in allocation, which are caused by deallocations happening at runtime by the `mltest` application – when the application frees the loaded model from memory and the CNode holding its physical page objects. These events of `mltest` runtime deallocation can also be identified as spikes in the alignment constraint fragmentation plot in Figure 4.13, happening around 1.3 MiB of requested memory for best fit and 2.7 MiB for next fit. Each physical page object for RISC-V 32-bit is 4096 bytes in size, and the CNode holding these objects is 8192 bytes in size. The drop in fragmentation

means that the allocator placed both the page objects of `mltest`'s model and its CNode on the same untyped slab in such a manner that the untyped slab's watermark needed to be aligned to 2-page (8192 bytes) size, wasting 4096 bytes of memory – the exact size of the both of the spikes. Of course, it is possible that this fragmentation is not introduced in the first place, if the page objects are allocated to a 2-page aligned address and the CNode is allocated in that slab, or if the CNode is allocated in a different slab. However, this event is a small confirmation that even though best and next fit do not attempt to place objects with the same lifetimes together, they still end up doing that in some cases (which is a good thing, as it minimises fragmentation [5]). Notably, this trace is an exception to hypothesis **H5**, stating that best fit allocator results in a higher number of untyped slab resets, and **H6**, which states that a higher number of untyped slab resets results in a lower fragmentation. It holds for the majority of the trace, but at around 2.7 MiB of requested memory, best fit allocator run is at 12 slab resets and 28.0 KiB total fragmentation, when next fit allocator run is at 18 and 43.9 KiB total fragmentation. Although surprising, it means that both hypotheses **H5** and **H6** cannot be taken as conjectures for all possible workloads, but as they hold for all other workloads explored in this work they shouldn't be completely disregarded.

The third trace is a random sample from the possible space of interleavings for 8 applications, each with 2 possible events, and its memory profile is presented in Figure 4.14. It is considered more challenging than the previous two, as it introduces more situations when memory is allocated for some applications and some is deallocated but never to a completely empty system state, leading to more possibilities of introducing watermarking constraint fragmentation. This indeed is visible for the next fit allocator, and tiny amounts of memory is also being lost due to watermarking constraint fragmentation for the best fit allocator.

Metric	**Sequence 1**		**Sequence 2**		**Sequence 3**	
Allocator:	best fit	next fit	best fit	next fit	best fit	next fit
Peak fragmentation (in KiB)						
watermarking:	0.00	0.00	**76.31**	212.25	48.50	**309.56**
alignment:	0.19	5.75	**28.38**	**46.88**	13.94	34.12
total:	0.19	5.75	**99.94**	230.69	62.00	**334.00**
Average fragmentation (in KiB)						
watermarking:	0.00	0.00	0.24	1.03	**1.01**	**63.99**
alignment:	0.00	4.81	**17.73**	**29.27**	10.80	20.88
total:	0.00	4.81	**17.96**	30.29	11.81	**84.87**

Table 4.5: Peak and average memory fragmentation for representative traces of sequential application interleavings. Total is computed as the peak/average of the sum of both fragmentation types (not the sum of the peak/average values). Highest value in each category is **bolded** for best and next fit.

Overall, traces of sequential application interleavings show that for these type of workloads best fit is far superior in terms of memory fragmentation (both due to the watermarking and alignment constraint). Table 4.5 shows fragmentation results for all three

workloads, similarly to the results for synthetic workloads, best fit allocator always results in a lower peak and average fragmentation[3] when compared to next fit allocator. This result was stated by hypothesis **H1** and further supports it. Further, for both traces 1 and 3 the number of untyped slabs was always lower for next fit allocator runs than for best fit allocator, but there was a single instance when this was not true for sequence 2. Although this means that hypothesis **H5** and **H6** do not always hold – it is not always the case that using best fit allocator results in more untyped slab resets, and it is not always the case that more untyped slab resets result in lower fragmentation, there is a lot of evidence that these two values are correlated. Further studies of the connection between these two phenomena are required to confirm this.

[3] Except for sequence 1, where watermarking fragmentation was never present for either allocators.

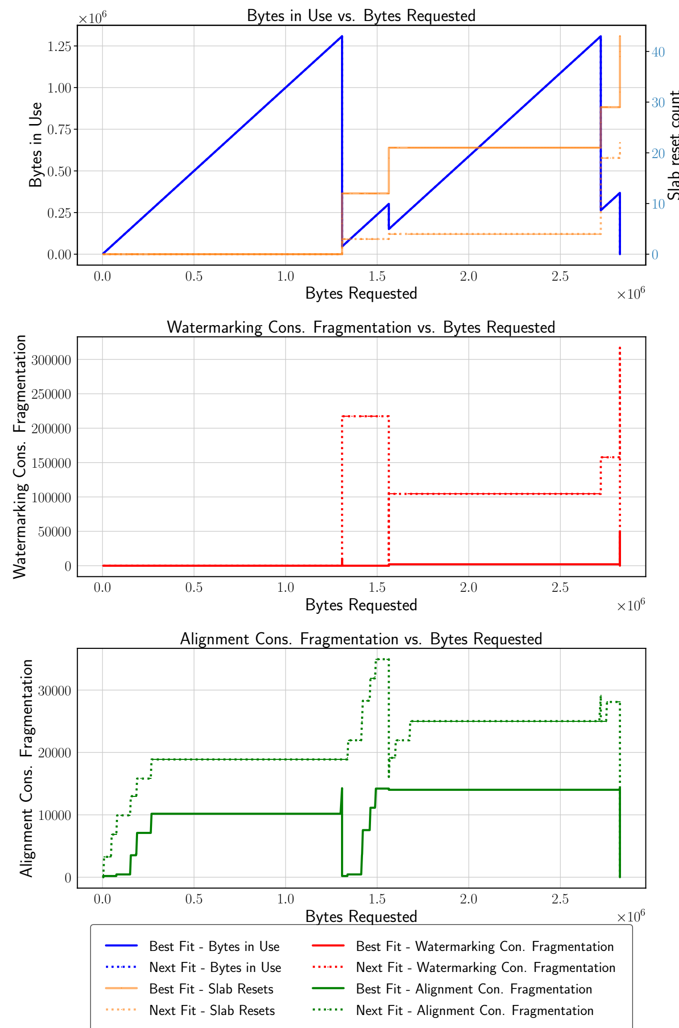


Figure 4.14: Memory profile of sequential app interleaving. Sequence 3: 3 apps started, 2 apps stopped, 3 apps started, 2 apps stopped, 2 apps started, 4 apps stopped.

4.2.4 Allocation Latency Evaluation

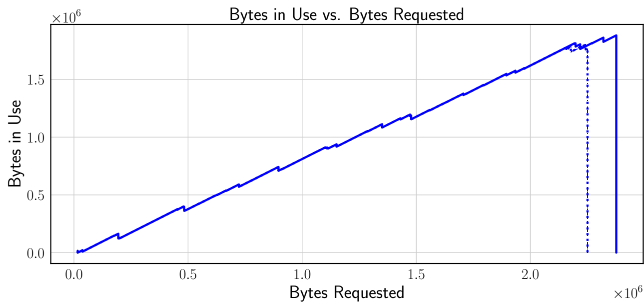
For measuring allocation latency, we used the exact same synthetic workloads as described in Section 4.1.2. Figures 4.15 and 4.16 show three plots each, similar to the earlier memory profile plots, showing results for both next-fit and best-fit allocator runs. The top plot displaying the bytes in use vs total bytes requested, representing the actual workload. The middle plot shows an instruction count vs bytes requested, the brown solid line being the instruction count per allocation for the best-fit allocator and the purple dashed line – for the next-fit allocator. Additionally, if out of memory errors occurred, they are marked on this plot as red "X"'s for best-fit, and red pluses for next-fit. For next-fit we additionally highlighted when `seL4_Untyped_Retype` invocations failed with orange circles, scaled in size with the amount of failed invocations. The bottom plot is a "control" plot, displaying the instruction count per free request (solid grey line for best-fit, dashed orange for next-fit). This plot is expected to be closely aligned for both allocators. Both them middle and the bottom plots are in log scale.

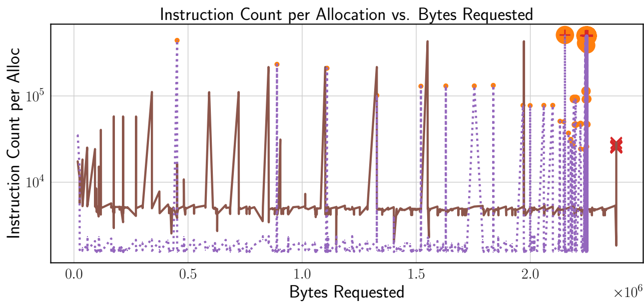
While analysing these results, we came to the unfortunate conclusion that there is some interference corrupting our results, causing large spikes in retired instruction counts. For next-fit these seem to positively correlate with failed untyped-retype invocations, which was expected (every spike in retired instruction count corresponds to at least one failed untyped-retype invocation). However, such spikes are also present for both presented best-fit runs. In order to try rand figure out the root cause of these spikes, we attempted exploring whether this is linked to either the type of requested object, or the number of slabs considered during allocation[4]. Unfortunately, none of these attempts uncovered the root cause of these spikes.

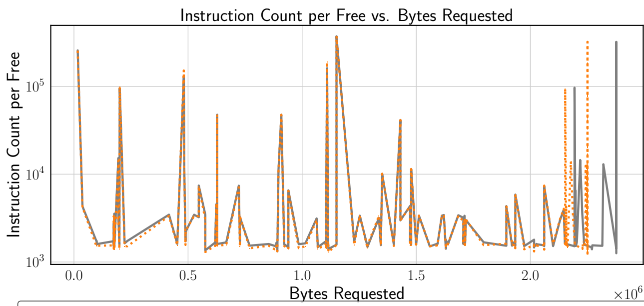
We therefore decided against drawing conclusions based on this dirty data to support or disprove our hypothesis, and propose that further studies to identify the cause of these spikes are necessary.

The unexplored possibilities behind these spikes include: pre-emption of allocation system calls introducing large delay in the kernel, pre-emption of allocation requests in the user-space (e.g. a different thread is getting scheduled while we are measuring the instruction count), or the untyped slab resets cause subsequent allocation requests to be much smaller.

[4]Best-fit performs an exhaustive search calculating number of left over bytes for each slab on every allocation, unless a perfect fit is found. We hoped that maybe these perfect fits are more often and the spikes only occur when all/majority of slabs are "visited".







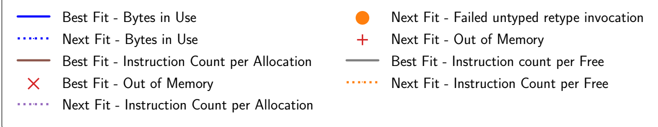
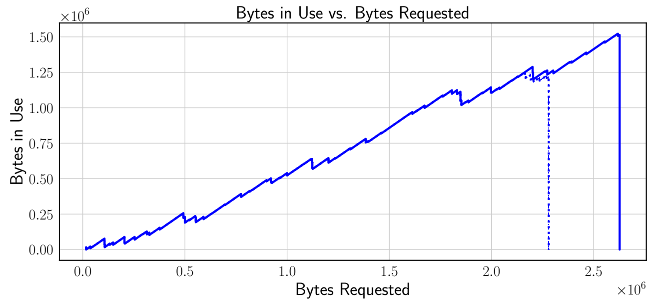
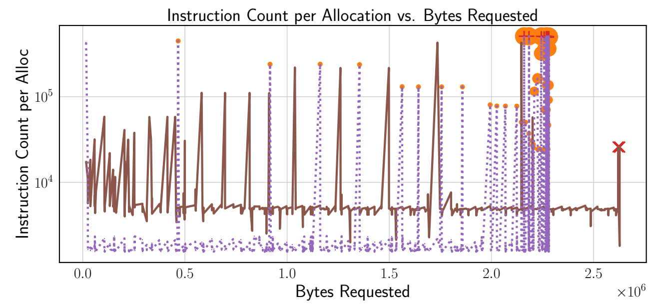


Figure 4.15: Latency of allocation and free requests for random uniform run. Seed 42, no. operations 1000, free chance: 16%.





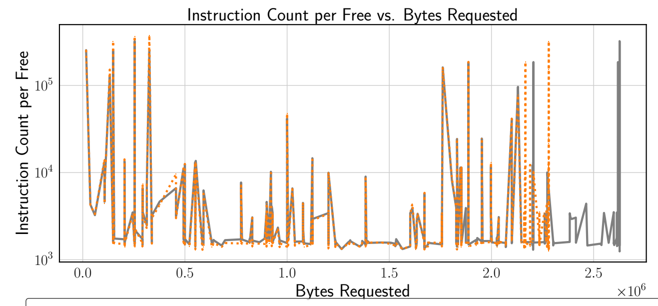


Figure 4.16: Latency of allocation and free requests for random uniform run. Seed 42, no. operations 1000, free chance: 32%.

Chapter 5

Conclusions & Future Work

5.1 Conclusions

We have successfully managed to validate or disprove the majority of the hypotheses stated in Section 4.1.4. Listed below:

H1. Best fit allocator produces smaller peak and average fragmentation than next fit. For all workloads both in the synthetic and representative category, hypothesis **H1** held.

H2. Untyped slabs fill up from smallest to largest for best fit, but in reverse order for next fit. The results from synthetic random workloads have shown that this hypothesis indeed holds.

H3. Peak instruction count-driven latency of best fit allocator is smaller than next fit and **H5. Use of best fit allocator results in higher number of untyped slab resets.** As we did not manage to uncover the cause of sudden increases in retired instruction counts for best-fit allocations, these two hypotheses remain unverified. Further studies in the directions proposed in Section 4.2.4 are required.

H5. Use of best fit allocator results in higher number of untyped slab resets and **H6. Higher number of untyped slab resets results in lower overall fragmentation.** For all synthetic workloads these hypotheses held, showing a clear correlation between number of untyped slab resets and lower overall fragmentation. For representative workloads, there was a single instance in time when next-fit allocator outperformed the best-fit allocator in terms of number of untyped slab resets.

We can confidently state that we have successfully met the goals of this project stated in Section 1.3:

Improving bookkeeping of memory in the CantripOS domain. The proposed modification to the seL4 microkernel and CantripOS operating system accurately tracks all memory allocations and deallocation in the system at a per-slab granularity. It has been successfully used to evaluate the performance of the original next-fit memory allocator and compare it against an implementation of best-fit memory allocator.

Eliminating failed allocation system calls when allocating memory in CantripOS. The implementation of best-fit memory allocator resulted in no failed untyped-

retype object invocations for all kinds of workloads. This could easily be extended for the original memory allocator.

Reducing memory fragmentation of the memory allocator of CantripOS. By implementing and evaluating the best-fit allocator against the original next-fit allocator, we have shown the hypothesis **H1** to be true. Therefore, along with the seL4 microkernel and CantripOS modifications, a best-fit allocator should result in lower memory fragmentation when deployed in the CantripOS system.

5.2 Future Work

Throughout the duration of this project, CantripOS and seL4 have proven to be exciting systems with incredible potential. We aim to continue working on this system, hopefully starting with upstreaming the results of this work to the public Open Se Cura repository. Future directions of this project should additionally include: reevaluating the latency of memory allocations, either by identifying the cause of "spikes" in retired instruction counts for best-fit allocator or by using a different metric when the target hardware of the system is more widely available. With access to hardware, the rest of the experiments could also be repeated to verify whether the emulated results are accurate. Furthermore, thanks to the contributions of this thesis, different memory allocators could be implemented and evaluated. For example, a targeted heuristic allocator tailored for seL4 and CantripOS could be designed and compared against the best-fit allocator.

Another interesting direction would be exploring how an allocator would behave with the watermarking constraint lifted, but all other constraints of seL4 and CantripOS enforced. This could be achieved, e.g., by creating a simulated allocator. seL4 previously had an experimental branch, which allowed for such memory allocation, but was never formally verified. If such a simulated allocator resulted in much lower fragmentation, maybe it would be worth revisiting this experimental branch.

Bibliography

[1] G. Heiser, “The seL4 microkernel – an introduction,” seL4 Foundation Whitepaper, May 2020. [Online]. Available: https://sel4.systems/About/seL4-whitepaper.pdf

[2] B. Randell, “A note on storage fragmentation and program segmentation,” *Commun. ACM*, vol. 12, no. 7, p. 365–ff., jul 1969. [Online]. Available: https://doi.org/10.1145/363156.363158

[3] G. Heiser, “How to (and how not to) use seL4 IPC,” microkerneldude.org, mar 2019. [Online]. Available: https://microkerneldude.org/2019/03/07/how-to-and-how-not-to-use-sel4-ipc/

[4] 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*, ser. IIES ’08. New York, NY, USA: Association for Computing Machinery, 2008, p. 35–40. [Online]. Available: https://doi.org/10.1145/1435458.1435465

[5] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, “Dynamic Storage Allocation: A Survey and Critical Review,” in *Proceedings of the International Workshop on Memory Management*, ser. IWMM ’95. Berlin, Heidelberg: Springer-Verlag, 1995, p. 1–116. [Online]. Available: https://dl.acm.org/doi/10.5555/645647.664690

[6] D. E. Knuth, *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997, pp. 435–452.

[7] T. A. Standish, *Data Structure Techniques*. USA: Addison-Wesley Longman Publishing Co., Inc., 1980, pp. 249–274.

[8] B. Karp, “Dynamic memory allocation in C,” UCL, COMP0019 Computer Systems, jan 2023, [PDF slides]. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/B.Karp/0019/s2023/lectures/0019-lecture6-mem-alloc.pdf

[9] M. S. Johnstone and P. R. Wilson, “The memory fragmentation problem: solved?” in *Proceedings of the 1st International Symposium on Memory Management*, ser. ISMM ’98. New York, NY, USA: Association for Computing Machinery, oct 1998, p. 26–36. [Online]. Available: https://doi.org/10.1145/286860.286864

[10] I. P. Page, "Optimal Fit of Arbitrary Sized Segments," *The Computer Journal*, vol. 25, no. 1, pp. 32–33, 02 1982. [Online]. Available: https://doi.org/10.1093/comjnl/25.1.32

[11] M. Gorman, *Understanding the Linux Virtual Memory Manager*. USA: Prentice Hall PTR, 2004, chapter: 6. [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/

[12] K. C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, no. 10, p. 623–624, oct 1965. [Online]. Available: https://doi.org/10.1145/365628.365655

[13] J. L. Peterson and T. A. Norman, "Buddy systems," *Commun. ACM*, vol. 20, no. 6, p. 421–431, jun 1977. [Online]. Available: https://doi.org/10.1145/359605.359626

[14] D. S. Hirschberg, "A class of dynamic memory allocation algorithms," *Commun. ACM*, vol. 16, no. 10, p. 615–618, oct 1973. [Online]. Available: https://doi.org/10.1145/362375.362392

[15] W. Burton, "A buddy system variation for disk storage allocation," *Commun. ACM*, vol. 19, no. 7, p. 416–417, jul 1976. [Online]. Available: https://doi.org/10.1145/360248.360259

[16] J. M. Robson, "An estimate of the store size necessary for dynamic storage allocation," *J. ACM*, vol. 18, no. 3, p. 416–423, jul 1971. [Online]. Available: https://doi.org/10.1145/321650.321658

[17] ——, "Bounds for some functions concerning dynamic storage allocation," *J. ACM*, vol. 21, no. 3, p. 491–499, jul 1974. [Online]. Available: https://doi.org/10.1145/321832.321846

[18] ——, "Worst case fragmentation of first fit and best fit storage allocation strategies," *The Computer Journal*, vol. 20, no. 3, pp. 242–244, jan 1977. [Online]. Available: https://doi.org/10.1093/comjnl/20.3.242

[19] J. E. Shore, "On the external storage fragmentation produced by first-fit and best-fit allocation strategies," *Commun. ACM*, vol. 18, no. 8, p. 433–440, aug 1975. [Online]. Available: https://doi.org/10.1145/360933.360949

[20] C. Lameter, "Slub: The unqueued slab allocator v6," LWN.net, mar 2007. [Online]. Available: https://lwn.net/Articles/229096/

[21] J. Bonwick, "The slab allocator: An Object-Caching kernel," in *USENIX Summer 1994 Technical Conference (USENIX Summer 1994 Technical Conference)*. Boston, MA: USENIX Association, Jun. 1994. [Online]. Available: https://www.usenix.org/conference/usenix-summer-1994-technical-conference/slab-allocator-object-caching-kernel

[22] J. Bonwick and J. Adams, "Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources," in *2001 USENIX Annual Technical Conference (USENIX ATC 01)*. Boston, MA: USENIX Association, Jun. 2001. [Online]. Available: https://www.usenix.org/conference/2001-usenix-annual-technical-conference/magazines-and-vmem-extending-slab-allocator-many

[23] 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*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[24] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, feb 2014. [Online]. Available: https://doi.org/10.1145/2560537

[25] Trustworthy Systems Team, Data61, "seL4 reference manual version 12.1.0," jun 2021. [Online]. Available: https://sel4.systems/Info/Docs/seL4-manual-12.1.0.pdf

[26] seL4 Foundation, "seL4 microkernel source code," GitHub repository, 2024, [file: RISC-V 32-bit constants.h]. [Online]. Available: https://github.com/seL4/seL4/blob/410b464c275c114cc70801e9aab11ea3eced21e9/libsel4/sel4_arch_include/riscv32/sel4/sel4_arch/constants.h#L20C1-L36C35

[27] S. J. Leffler, J. Tate-Gans, and Scott, "Announcing KataOS and Sparrow," oct 2022. [Online]. Available: https://opensource.googleblog.com/2022/10/announcing-kataos-and-sparrow.html

[28] K. Yick, "Project Open Se Cura Open Source Announcement," nov 2023. [Online]. Available: https://opensource.googleblog.com/2023/11/project-open-se-cura-open-source-announcement.html

[29] Antmicro, *Renode version 1.15 manual*, 2024. [Online]. Available: https://renode.readthedocs.io/en/latest/

[30] Google Open Source, "Open Se Cura Project source code," Git repositories, 2024. [Online]. Available: https://opensecura.googlesource.com

[31] The Rust Programming Language, "The Rustonomicon," may 2024, [Book is work-in-progress]. [Online]. Available: https://doc.rust-lang.org/stable/nomicon/ownership.html

[32] seL4 Foundation, "camkes-tool source code," GitHub repository fork, 2024, [Note: repository not anonymised]. [Online]. Available: https://github.com/Willmish/camkes-tool/tree/willmish/visualCAmkES_bump_python3

[33] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "Camkes: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, 2007, Component-Based Software Engineering of Trustworthy Embedded Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412120600224X

[34] seL4 Foundation, "rust-sel4 source code," GitHub repository, 2024. [Online]. Available: https://github.com/seL4/rust-sel4

[35] ——, "seL4 microkernel source code," GitHub repository, 2024. [Online]. Available: https://github.com/seL4/seL4

[36] J. Liedtke, "On μ-kernel construction," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 237–250. [Online]. Available: https://doi.org/10.1145/224056.224075

[37] The Rust Programming Language, "smallvec crate," 2024. [Online]. Available: https://docs.rs/smallvec/latest/smallvec

[38] S. J. Leffler, personal communication, nov 2023.

[39] Google Open Source, "Open Se Cura Project source code," Git repositories, 2024, [file: sel4.xml]. [Online]. Available: https://opensecura.googlesource.com/3p/sel4/sel4/+/d860c42ca870694e6d5ae208b0bf762093a5a014/libsel4/include/interfaces/sel4.xml#53

[40] seL4 Foundation, "seL4 Tutorial: Capabilities," 2024. [Online]. Available: https://docs.sel4.systems/Tutorials/capabilities.html#cnodes-and-cslots

[41] Kent McLeod, personal communication, feb 2024.

[42] seL4 Foundation, "sel4test: documentation," 2024. [Online]. Available: https://docs.sel4.systems/projects/sel4test/

[43] B. Zorn and D. Grunwald, "Evaluating models of memory allocation," *ACM Trans. Model. Comput. Simul.*, vol. 4, no. 1, p. 107–131, jan 1994. [Online]. Available: https://doi.org/10.1145/174619.174624

[44] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA version 2.1," Electrical Engineering and Computer Sciences, University of California, Berkeley, Tech. Rep. EECS-2016-118, May 2016. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf

Appendix A

Code of the Modified CantripOS and seL4 System

A.1 Code for all repositories used in the project

Note, the below repositories are not anonymised and contain both the name and surname of the author. All URLs are for the `github.com` service.

Modified CantripOS system with code for performance analysis, for the improved design (Section 3.3):

1. CantripOS: https://github.com/Willmish/cantrip/tree/willmish/performance_eval_O1_mod,
2. seL4: https://github.com/Willmish/seL4/tree/willmish/memory_management_O1_mod,
3. sim-tests (Renode and Robot Framework scripts for scheduling traces): Willmish/sim-tests/tree/willmish/performance_eval_O1_mod,
4. sel4test harness: Willmish/sel4test/tree/willmish/memory_management,
5. capdl (Capability Distribution Language): Willmish/capdl/tree/willmish/memory_management,
6. seL4 Libraries: Willmish/sel4_libs/tree/willmish/memory_management.

Code for the first attempt of improved design, only the repositories which have large changes (Section 3.2):

1. CantripOS: Willmish/cantrip/tree/willmish/memory_management,
2. seL4: Willmish/seL4/tree/willmish/memory_management.

Code for the performance evaluation of memory allocator logs, and miscellaneous:

1. Python scripts for evaluation: Willmish/cantrip_memory_allocators_eval,
2. camkes-tool: Willmish/camkes-tool/tree/willmish/visualCAmkES_bump_python3.

Appendix B

Memory System Redesign: Fix for dirty untyped object splitting

B.1 Description and bugfix of the dirty untyped object splitting bug

When the total size of the system is below the halfway point of an untyped slab on which it is being allocated, the original implementation splits the slab in half, and “fills up” the smaller half with various sized untyped objects to minimise wastage, and allocates a large slab for the remaining half, just like the Figure B.1 shows. However, if the system exceeds the halfway point, the original system implementation did not take this into account and gets stuck in an endless loop. Modifying the logic to distinguish when system components are below or above the halfway point and only perform the “filling up” logic when beyond the halfway point, as presented in the lower diagram in Figure B.1, resolves the issue.

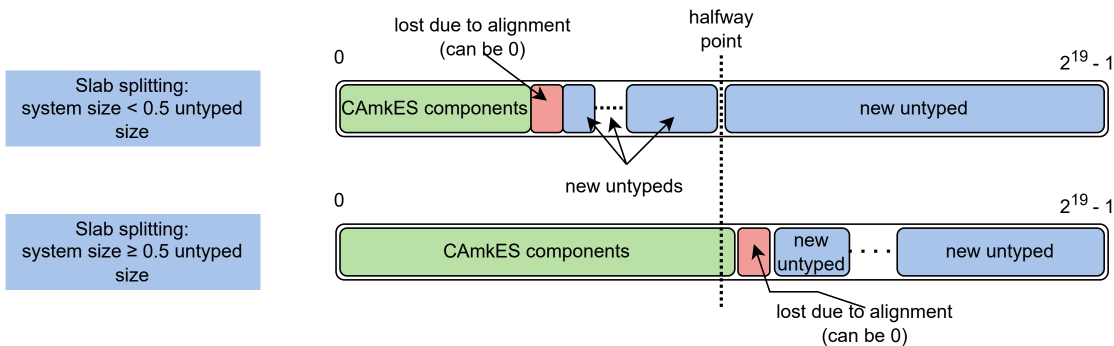


Figure B.1: State of the slab containing the system components after splitting. Top figure shows an example state when: size of components $<$ 0.5 size of the slab. The bottom figure shows an example state for the reverse inequality.

Appendix C

Memory System Redesign: Neighbour Traversal Inefficiency

C.1 Empirical confirmation of neighbour traversal inefficiency

To verify whether the original attempt at system redesign actually introduces an additional `O(n)` complexity during `seL4_CNode_Delete` invocation, I performed an experiment of allocating 50 frame objects on a single untyped slab, and subsequently freed each one in the order of allocation as well as in the reverse order, logging the length of neighbour traversal in the Mapping Database linked list.

As expected, when freeing in the same order as allocation order, on each deallocation there are initially 50 traversals, decreasing by 1 after each free, showing that indeed there is a worst-case `O(n)` complexity introduced. Doing this in reverse, however, always results in a traversal of length 1.

Introduced command for traversal in user space: 3b04aee and logging in seL4 kernel 4a52639. Logs of results:

```
Slab:[59, bits 20] available 843776, max_size_bytes 1048576,
    allocated_bytes: 204800, allocated_objects: 50
Performed: 50 traversal steps during delete
Free'd ObjDescBundle { cnode: 1, depth: 32, objs: [ObjDesc { type_:
    seL4_RISCV_4K_Page, count: 1, cptr: 24 }] }
Performed: 49 traversal steps during delete
Free'd ObjDescBundle { cnode: 1, depth: 32, objs: [ObjDesc { type_:
    seL4_RISCV_4K_Page, count: 1, cptr: 25 }] }

```

Appendix D

Additional synthetic workload results

D.1 Memory statistic for 32 and 64 percentage chance for deallocation runs.

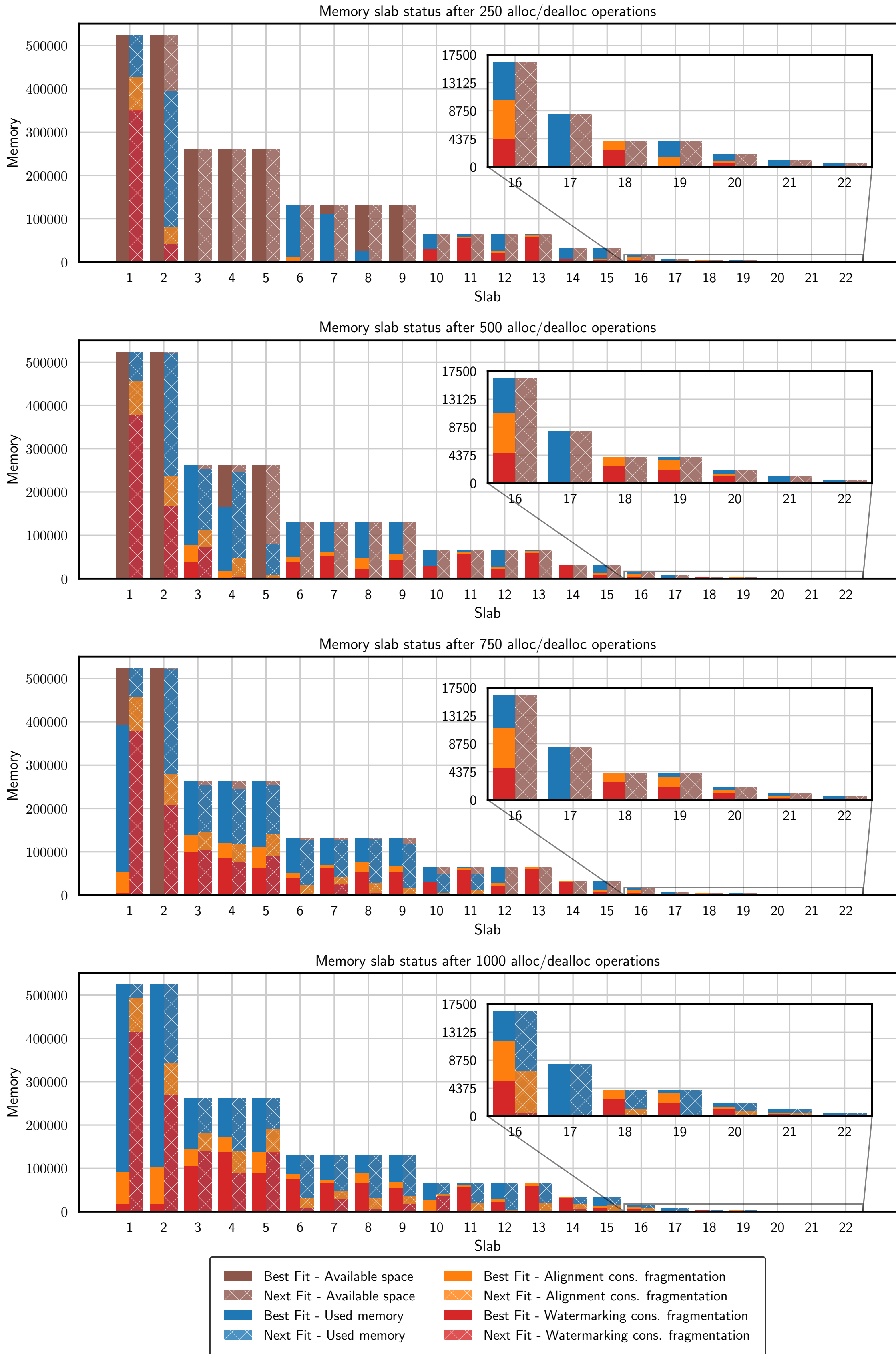


Figure D.1: Per-slab statistics for a random uniform run. Seed: 42, no. operations: 1000, free chance: 32%.

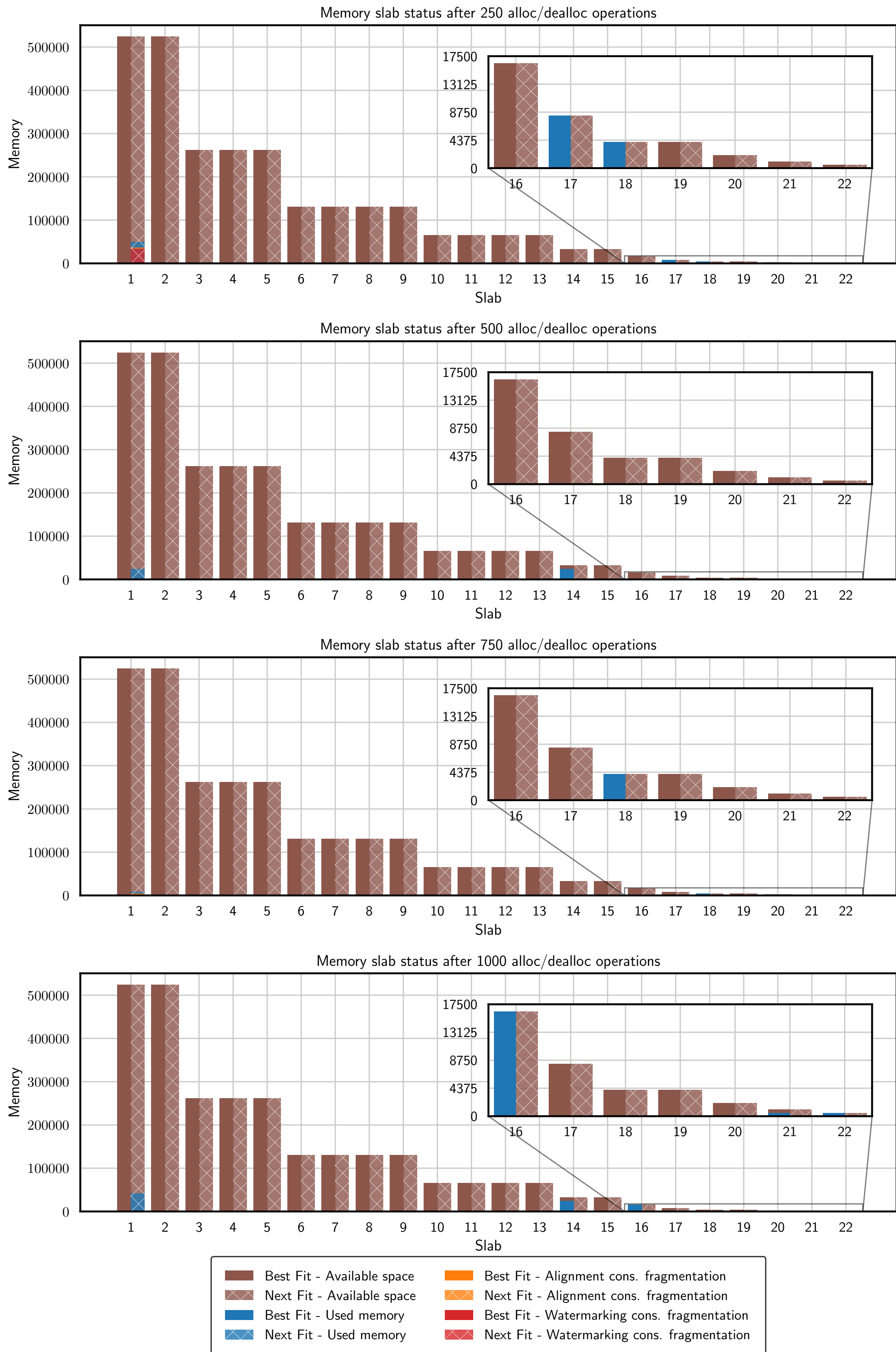


Figure D.2: Per-slab statistics for a random uniform run. Seed: 42, no. operations: 1000, free chance: 64%.

Appendix E

Project Plan

Project Plan

Project Title: Improving the Space Efficiency of the Memory Management System of CantripOS

Candidate Number: FMVC9
MEng Computer Science
Supervisor: Prof. Brad Karp

Submission date: 19 November 2023*

1 Aims and Objectives

The aims and objectives of the project are outlined below.
Aim: To learn more about the design of memory management systems by experimentally studying the space and compute efficiency of the memory allocation and management system of CantripOS and improving this system's space efficiency. Because CantripOS uses Rust (a memory safe language) and seL4 secure microkernel, this project will advance my understanding of using memory safe languages for system implementation and secure microkernels.
Objectives:

1. CantripOS is an experimental operating system and targets hardware that is not available to me, therefore a first step to work with this experimental system is to set up the development environment.
2. Next step involves implementing support for bookkeeping in memory allocation. This experimental OS has a rudimentary allocator system - it performs first fit allocation which is known to not be space efficient. It also lacks bookkeeping information about allocated memory in the application layer, which limits the performance of the memory allocator and the quality of allocation.
   (a) Memory management in CantripOS has 2 layers - an application layer (CantripOS) through which apps make requests for memory allocation and seL4 microkernel to which the requests are made to. To improve the current memory management, I need to extend the interface between the user level memory allocator and the seL4 microkernel, so that the user level allocator can have more knowledge of what memory is currently free or allocated.
   (b) Using the extended interface with seL4 microkernel, I will implement the user level memory bookkeeping system.

*This is a supervisor agreed revision on the original submission from 19.11.2023

3. Previously known efficient memory allocators include: best fit - which is known to minimise memory waste but can be computationally expensive, as well as other variants of best fit - which reduce computational cost but are more memory wasteful. Preferred goal is to implement 2 such algorithms for CantripOS allocator, but depending on time restrictions I might implement both or only one.

Aim: To evaluate and compare the old and new memory management methods based on the metrics of: efficiency in memory utilisation (less memory wasted is better) and computational overhead (less computation per allocation is better).
Objectives:

1. Identify and implement applications of synthetic workloads for worst and best cases of the old and new memory allocation algorithms. These workloads would have human-engineered patterns of allocation, replicating the spectrum of different memory allocation behaviours, e.g. making many big allocations of large arrays, making many small allocations, doing a lot of deallocations.

2. Identify and attempt porting a few applications that are representative of what users might expect to run on the CantripOS system and the hardware it targets.

3. Perform experiments on the old and new memory allocation system using the applications described above, measure their performance in terms of computational overhead of allocation and freeing, and memory utilisation and fragmentation.

2 Expected Outcomes and Deliverables

This project aims to produce the following results (roughly in chronological order):

1. A Project Plan outlining the aims of the proposed project. (This document)

2. An Interim Report (By 19th January 2024), outlining the current progress of the project, including what work is needed to complete the project, along with the time needed and milestones for completion.

3. Source code of modified seL4 microkernel along with the CantripOS Operating System with proper bookkeeping system of memory in CantripOS.

4. Source code for the memory allocator(s) for CantripOS, implementing an algorithm(s) with expected improved efficiency of memory management.

5. Pseudo-code and source code for example CantripOS applications with synthetic workloads for worst and best cases of the old and new memory allocation algorithms in terms of memory waste and computational efficiency.

6. Identifying the applications representative of what users might expect to run on the CantripOS system and the hardware it targets.

7. Source code for any porting effort of the representative applications to the CantripOS platform.

8. Source code for a script used for collecting benchmarking data from running applications, and performing experiments using these applications - e.g. automatically executing the applications over UART and saving the data from UART output.
9. Results of the experiments and analysis of the memory management system using the previously written applications, measuring the computational overhead, memory utilisation and fragmentation of memory allocation and freeing.
10. Extensions: enhancing the memory allocator to support debugging applications using debbuging memory allocator, implementing memory allocation algorithm using exponentially sized bins, adding visual representation of allocated memory (for easier spotting of fragmentation), upstreaming attempt of the source code.
11. Submitted Final Report by 4pm on 26th April 2024.

3 Work Plan

With both development and performance analysis playing a big role in the project's final outcome and success, the work plan is divided into two halves - first one being fairly focused on the software development, and the second one on performance experiment planning, running and analysis.

- Project Start - End of October (4 weeks) : Exploration of project direction possibilities, background reading, setting up the development environment for CantripOS and seL4.
- November - Mid December (6 weeks) : Getting in touch with Open Source project maintainer, Setting up the development environment, following the suggested leads - extending the interface between user level CantripOS and seL4 with memory allocation/freeing information, implementing bookkeeping for memory management in CantripOS.
- Mid December - January (2 weeks) : Writing the report section for the code design and development part. (At the same time as the code is being developed)
- Mid December - January (2 weeks) : Implementing the improved memory allocation system in CantripOS (at least 1 algorithm).
- January - Mid January (2 weeks) : Identifying and implementing synthetic workload applications for CantripOS's new and old memory system.
- Mid January - February (2 weeks) : Identifying and porting applications representative of typical workloads on CantripOS.
- February - End of February (3 weeks) : Performing experiments on CantripOS and writing preliminary analysis.
- End of February - End of March (4 weeks) : Finalising analysis, adding any extensions to the memory allocation system, running additional experiments if applicable, upstreaming to Open Source if successfully improved efficiency.
- End of March - Submission (4 weeks) : Finalising the Final Report and iterating with the supervisor for final submission.

Appendix F

Interim Report

Project Interim Report

Project Title: Improving the Space Efficiency of the Memory Management System of CantripOS

FMVC9
MEng Computer Science
Supervisor: Brad Karp

Submission date: 2 February 2024

1 Progress to Date

Majority of the work done up to this point was spent on improving understanding of the kernel (seL4) along with, novel to me, safety mechanisms implemented in the kernel.

During the months of October and November I got the existing code base of the CantripOS project and set up the emulation environment (as the target hardware for this system is not widely available yet). I have spent some time trying to set up the emulation and build environment on a laptop with a different CPU architecture (M1) which unfortunately turned out to not be feasible.

During December I worked through the majority of tutorials available in seL4 documentation as well as explored the documentation introducing the concepts of Capabilities (pointers with privilleges), Untypeds (how free memory is represented in seL4).

From January up to now I have traced (both in code and through execution) the memory initialisation in seL4 and how free memory blocks are created and handed off to userspace. I instrumented the kernel to track memory allocation.

I have explored how to use the GDB debugger for both the kernel and userspace and am currently in the process of tracing how memory allocation is currently handled in CantripOS and what information is kept about the free and used memory. Ideally after completing this stage I will have an understanding of what information is needed from the kernel to improve the memory management system, so I can design and implement an extension to kernel's interface with the userspace.

Learning curve on the design and implementation of both seL4 and CantripOS is higher than anticipated, and although I was hoping to be developing the system modifications at this point in time, I am still at a point where I am refining my understanding of the seL4 kernel and CantripOS system. With my supervisor I will be speaking directly to the author of CantripOS to accelerate the process of understanding CantripOS. Below is a revised plan which takes into account the high learning curve of the project.

2 Remaining Work

The majority of the work that needs to be done, includes: improving understanding of current memory management system in CantripOS, design and implement kernel-userspace interface modification, design and implement the improved userspace memory allocation system, prepare synthetic workload applications and perform experiments on CantripOS and finally identify and port applications representative of typical workloads on the system.

The revised timeline for the project is outlined below:

1. 24th Jan - 7th Feb (2 weeks): Building an understanding of the memory management system in CantripOS by tracing its memory allocator. This is required to understand what is lacking in the current book keeping of memory and how the interface with the kernel space (seL4) needs to be modified to allow improving it.

2. 7th February - 16th Feb (1.5 weeks): Prototype/design on paper the required modification of the kernel and userspace. Document progress up to date in the Final Report.

3. 16th Feb - 28 Feb (1.5 weeks): Implement and test kernel interface modification along with the improved memory allocation system.

4. 28 Feb - 6 Mar (1 week): Design and implement synthetic workloads for CantripOS's new and old memory system.

5. 6 Mar - 13 Mar (1 week): Reach out to the CantripOS project group and identify applications representative of typical workloads on CantripOS. Port the application. Document the modification to the memory system along with the implemented synthetic workloads in the Final Report.

6. 13 Mar - 27 Mar (2 weeks): Performing experiments on CantripOS and writing preliminary analysis in the Final Report. Ironing any potential bugs discovered in the memory allocator implementation.

7. 27 Mar - 10 April (2 weeks): Finalising analysis in the Final Report, adding any extensions to the memory allocation system, running additional experiments if applicable, upstreaming to Open Source if successfully improved efficiency.

8. 10 April - Submission (2.5 weeks): Finalising the Final Report and iterating with the supervisor for final submission.