Karlsruhe Institute of Technology

# Toward Specialization of Memory Management in Unikernels

Bachelor's Thesis
submitted by

cand. inform. Hugo Lefeuvre

to the KIT Department of Informatics

| | |
|---|---|
| Reviewer: | Prof. Dr. Frank Bellosa |
| Second Reviewer: | Prof. Dr. Wolfgang Karl |
| Advisor: | Dr.-Ing. Marc Rittinghaus |

01. January 2020 – 03. June 2020

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, June 3, 2020

# Abstract

Unikernels are standalone, single-purpose appliances assembled from user-space source code at compile time. Unikernels achieve high performance, low memory footprint, and fast boot time with a small trusted computing base. These characteristics are obtained by specializing operating system components toward the application at build time. Amongst the classical system components, the dynamic memory allocator is known to have a crucial impact on performance and its specialization potential is well recognized in the literature. In particular, it is known that allocators typically trade performance for security or memory usage for performance, and no general-purpose allocator will behave perfectly in all situations. Despite of this, unikernels do not typically offer more than a single general-purpose memory allocator.

This thesis explores the idea of leveraging memory allocators as a specialization layer in unikernels. We analyze dynamic memory allocation in unikernels, and how it can be specialized. Following this, we design, implement, and evaluate allocator specialization in Unikraft, a unikernel framework. To this end, we port several general-purpose memory allocators to Unikraft and measure the performance of off-the-shelf applications across a wide range of scenarios and key performance indicators. By specializing the allocator, we reduce the boot time of Unikraft from 7.4 ms to less than 1.5 ms for an Nginx unikernel. We show that allocator specialization can improve performance and memory footprint at moderate cost, achieving speedups of up to 12% for Redis and 26% for SQLite and reducing the memory footprint by 22% for Redis and 50% for Nginx.

# Acknowledgments

I would like to express my gratitude to my supervisor, Dr. Marc Rittinghaus, for guiding me through this journey and having the patience of reviewing my work. This thesis would have been very different without his insightful comments.

I would like to thank my colleagues at NEC Laboratories Europe, Simon Kuenzer, Dr. Felipe Huici, Alexander Jung, Sharan Santhanam, and Dr. Roberto Bifulco for letting me take part to their research. I learned a lot from them. I am grateful to NEC Laboratories Europe GmbH for funding this thesis and allowing me to work with such a competent team.

Finally, I would like to thank my family, for their support throughout my studies, for the pictures of bread and cats that warmed my heart in the time of Corona.

# Contents

# Chapter 1

# Introduction

Unikernels are standalone, single-purpose appliances assembled from user-space source code at compile time. Unikernels achieve high performance, low memory footprint, and fast boot time with a small trusted computing base [64]. These characteristics are obtained by specializing operating system components toward the application at build time. Freed from the constraints of general-purpose operating systems, unikernel components can be specialized to enable, for instance, fast boot time and flexible deployment of cloud services (e.g., just-in-time instanciation [65]), low memory footprint and high density (e.g., high density TLS termination [66]), or runtime guarantees for real-time systems [25].

Amongst the classical system components, the dynamic memory allocator is known to have a crucial impact on performance and its specialization potential is well recognized in the literature [14]. In particular, it is known that memory allocators typically trade performance for security or memory usage for performance and no general-purpose allocator will behave perfectly in all situations [91]. Despite of this, most well-known unikernels do not offer more than a single dynamic memory allocator.

This thesis explores the idea of using memory allocators as a specialization layer. We investigate the benefits and costs bound to different classes of allocators and investigate whether and how these can be leveraged to optimize toward different key performance indicators (KPIs) such as image size, boot time, runtime performance, memory footprint, or security.

Most of the previous research on dynamic memory allocation has been realized in the context of general-purpose operating systems. Nevertheless unikernels and general-purpose operating systems present significant differences with regard to dynamic memory allocation: unikernels run as a single process, in a single processor mode, with a single address space, making expensive system calls as cheap as simple procedure calls [52]. This thesis investigates the transferability of previous research to unikernel environments.

The first part of this thesis analyzes the specialization potential of memory allocators in unikernels. We discuss the differences between unikernels and general-purpose operating systems with regard to memory allocation, how these differences affect the design of memory allocators in unikernels, and therefore how they impact allocator specialization strategies. Then, we propose a concrete allocator specialization approach for Unikraft [56], a unikernel framework featuring an extensive configuration and build system.

Following this, we evaluate our approach by porting several allocators to Unikraft and evaluating the performance of these allocators across a wide range of scenarios with off-the-shelf applications. We measure the performance gain that can be obtained by specializing the allocator from the perspective of four KPIs: image size, boot time, runtime performance, and runtime memory usage. In particular, this thesis makes the following contributions:

- We provide a systematic analysis of dynamic memory allocation in unikernels. In doing so, we analyze the impact of the cost of system calls (and more specifically mode switch costs) on memory allocation.

- We port several memory allocators to the Unikraft framework, adapting and improving the existing API, and document the technical difficulties that can arise when porting a user-space allocator to be used as kernel allocator.

- We show that memory allocators can significantly influence the boot time of unikernels. By specializing the allocator, we achieve minimal boot times of 39 µs for a noop kernel, 113 µs for an SQLite unikernel, and 1.4 ms for Nginx and Redis unikernels.

- We show that allocator specialization can improve performance and memory footprint at reasonable cost in unikernels, achieving speedups of 12% and 26% for Redis and SQLite and reducing Redis' and Nginx's memory footprint by 22% and 50%.

This thesis is structured as follows: Chapter 2 provides background to unikernels, library operating systems, and dynamic memory allocation. The architecture and characteristics of Unikraft are presented, along with the memory allocators that are studied throughout this thesis. Chapter 3 analyzes the differences between general-purpose operating systems and unikernels with regard to dynamic memory allocation and investigates the specialization potential of memory allocators in the context of unikernels. Chapter 4 presents the porting process of the different allocators in Unikraft and the challenges involved. Chapter 5 presents our measurements and discusses our results. Finally, Chapter 6 concludes and offers insights into future research.

# Chapter 2

# Background and Related Work

This thesis explores the idea of leveraging memory allocators as a specialization layer in unikernels, and reviews the differences between general-purpose operating systems (OS) and unikernels with regard to dynamic memory allocation. The following chapter provides relevant background for the rest of this thesis.

Section 2.1 focuses on library operating systems and unikernels. We introduce unikernels and explain how they differ from general-purpose operating systems and containers. We present Unikraft and compare it with other unikernel approaches in the literature, effectively motivating the choice of Unikraft for this thesis. Section 2.2 provides background to dynamic memory allocation. We introduce the memory allocation problem and discuss custom and general-purpose memory allocation. Finally we present four memory allocators that will be studied as part of this work: Unikraft's buddy allocator, Mimalloc, TLSF and tinyalloc.

## 2.1 Unikernels

Cloud services offer networked access to a large pool of computing resources, accessible by customers on a highly flexible basis. The cloud backend is subject to high stress: services can be scaled up and down at any moment, and high availability is expected [27]. As a consequence there is a surge to develop efficient resource pooling and virtualization technologies.

Virtual Machines (VMs) are hardware abstractions that can be used by cloud providers to emulate several self-contained machines on a single physical server. The VM abstraction is provided by a hypervisor that sits directly on the hardware, allowing VMs to be created, suspended, resized, and migrated to other physical hosts dynamically. Virtual machines running a general-purpose operating system such as Linux offer customers an environment that resembles a physical machine, with the added flexibility of cloud technologies.

While classical VMs seemingly provide a pertinent answer to the cloud computing problem, they are also associated with slow boot times, insufficient density and scalability [83]. In fact, general-purpose operating systems have not been designed to run in virtualized cloud environments. The process abstraction overlaps with the VM abstraction in a cloud industry vastly relying on single-purpose appliances [52]. In addition, the hardware abstraction offered by the OS overlaps with the hardware abstraction already done by the hypervisor [52].

These observations led to an increased resort to lightweight virtualization. Instead of virtualizing an entire machine, lightweight virtualization technologies such as containers virtualize the operating system only [83]. Containers rely on OS facilities to provide abstraction and isolation. In Linux, lightweight virtualization can be implemented using *control groups* (cgroups) [42] or *namespaces* [51]. Control groups allow to define groups of processes whose usage of system resources can be limited and monitored. Namespaces provide an additional layer of abstraction over system resources; each namespace provides the illusion of an isolated instance of the resource managed by the operating system.

Containers achieve performance comparable to bare metal, with fast startup and instanciation time [83]. They allow for dense systems: up to thousands of containers per machine, while virtual machines rarely manage more than hundreds [66]. Unfortunately, the performance of lightweight virtualization comes at the cost of security. General-purpose operating systems — upon which containers rely — expose a trusted computing base (TCB) typically tens of times larger than hypervisors, resulting in increased attack surface [37]. Linux's system call application programming interface (API) has known an uninterrupted growth since the beginning of the 2000s [66]. Security vulnerabilities not only in the cgroup and namespace subsystems but in any part of the kernel could allow applications to interfere across container boundaries. Comparatively, VMs running on the same hypervisor would need to exploit a vulnerability in the hypervisor itself to interfere with each other, which is less likely due to the smaller TCB of hypervisors. The isolation offered by lightweight virtualization technologies is therefore much more brittle.

In conclusion, classical virtual machines appear burdened by an excess of abstraction. Containers address the problem of performance but suffer from an insufficient degree of isolation. This motivates the need for a trade-off that would combine performance and isolation, effectively inspiring the development of *unikernels* in the course of the 2010s.

### 2.1.1 Introduction to Unikernels

The *unikernel* approach to virtual machines was used for the first time by Madhavapeddy et al. [64] as part of their work on Mirage, a novel approach to deploy

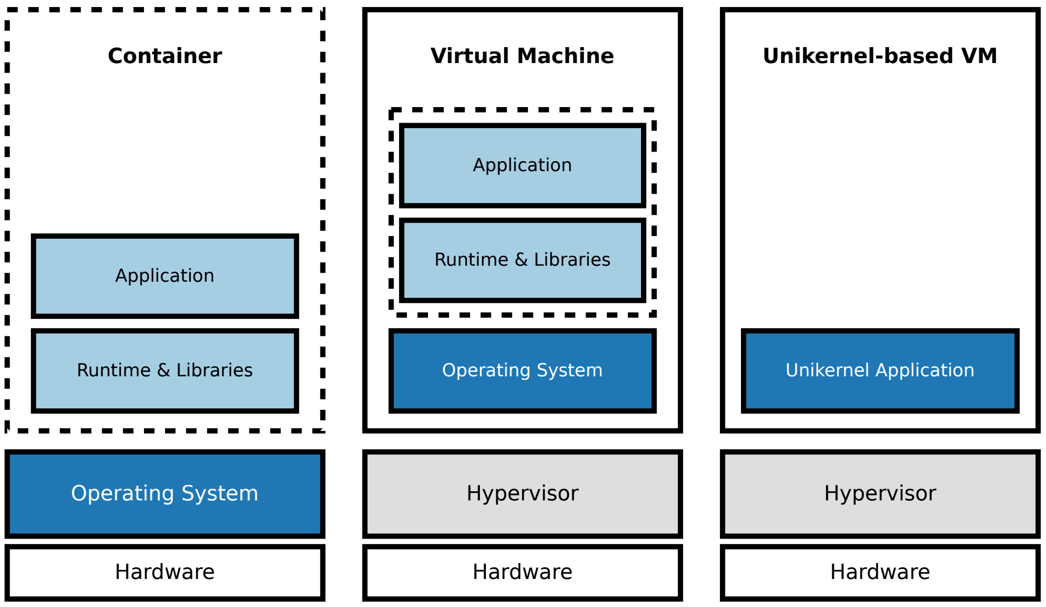


Figure 2.1: Comparison of containers, general-purpose and unikernel VMs [38]. Containers run a user-space stack on top of a shared kernel, resulting in high performance but low isolation. General-purpose VMs run a full OS stack on top of a hypervisor, resulting in high isolation but low performance. Unikernel-based VMs run a single-image, specialized OS stack on top of a hypervisor, resulting in high isolation and higher performance than general-purpose VMs.

cloud services written in OCaml. Unikernels are defined as single-purpose, self-contained appliances built from user-space source code at compile time. In the unikernel approach, OS services and abstractions, language runtime, libraries, and application code are specialized, compiled, and linked together as a single kernel image that can run in virtualized environments. Both kernel and application thus run as a single process in privileged mode with a single address space, effectively removing the distinction between user and kernel-space. The fundamental idea behind unikernels is that by sacrificing the versatility of general-purpose virtual machines, unikernel-based VMs can achieve performance closer to that of containers without shelving isolation [66]. Figure 2.1 offers a comparative overview of containers, general-purpose and unikernel VMs.

Unlike containers, unikernels are self-contained. Unikernels rely on the VM abstraction offered by the hypervisor and implement all device drivers, operating system services, and abstractions required by the application to work. Containers, on the other hand, rely on a shared kernel and only virtualize the operating system environment. Unikernels are therefore significantly more isolated.

Containers typically stand out through their fast startup time and high density. It was shown that unikernels can also provide excellent numbers for these performance indicators, as a result of their lightweight code base [66, 73]. However, this performance is generally tainted by bottlenecks at the hypervisor level. Xavier et al. [93] found an average 4-7x startup time slowdown essentially due to VM instanciation overhead in the hypervisor. On the other hand, Manco et al. [66] show that this overhead can be avoided using a custom hypervisor toolstack, making unikernel VMs boot as fast as `fork()` or `exec()` system calls. More recent work by Olivier et al. [73] show that unikernels coupled with custom hypervisor toolstacks typically achieve boot and destruction times as fast or faster than containers.

Unlike general-purpose VMs, unikernels run entirely in privileged mode, with a single address space. This allows unikernels to take a liberal approach to hardware abstraction which is often criticized in general-purpose OSs. A compelling example is asynchronous I/O. Enberg et al. [31] point out that this abstraction was developed at a time when I/O speed was much slower than the CPU: a program would asynchronously request an I/O operation and be interrupted when the operation succeeded. Now, the overhead imposed by the operating system's I/O abstraction and the following asynchronous interruption is comparable to the time actually taken by the hardware to complete the I/O task. Unikernels can achieve performance gains by eschewing such abstractions.

Finally, in contrast to general-purpose VMs which typically incorporate a kernel, multiple libraries, language runtimes, and applications, unikernels are single-image systems. The entire software stack is configured, compiled, and linked as a single bootable binary, allowing for a high degree of specialization and optimization. This high degree of specialization and optimization is an essential factor in the performance of unikernels [68].

### Specialization and Optimization

Unikernels are designed to serve the purpose of a single application. There is as such no need for all operating system services, abstractions, and interfaces to be present in the unikernel image. Unused code is avoided, as it increases the trusted code base and slows down boot time by unnecessarily initializing unused functionalities. Unikernels are therefore developed as a set of libraries that independently implement particular operating system services[1] . Unikernels

[1]Unikernels can thus be seen as an approach to library operating systems [64]. Classical library operating systems such as Exokernel [32] (1995) or Drawbridge [76] (2011) relocate the so-called *OS personality* of monolithic operating systems to user-space, within the process. The OS personality is defined as the implementation of operating system APIs and application visible semantics [76]. This approach to library operating systems is now commonly referred to as *pico-*

can be specialized by opting out irrelevant services and abstractions, so that only necessary components are embedded in the final image.

Along with image size reduction, certain unikernels allow for per-component specialization. Operating system components such as the network stack can be optimally configured with regard to application and environment specific constraints. In addition, provided that the unikernel offers multiple implementations of the same component (e.g., as selectable libraries), the most appropriate implementation can be chosen depending on the application and targeted key performance indicators (KPIs) [56].

Finally, the compilation and linking process allows for a consistent, system-wide optimization policy driven by targeted KPIs such as performance or space. The unikernel linker has a comprehensive overview of the system and can perform aggressive link time optimizations (LTO) such as dead code elimination [64] and function inlining.

### Unikernel Approaches

It is possible to classify previous approaches to unikernels in the literature into three categories. First, unikernels that are fully hand-crafted for a particular application, such as ClickOS [68], an OS optimized for network function virtualization. Hand-crafted unikernels offer optimal performance as a result of their extreme specialization, however, the porting time is maximal. Second, unikernels that specialize on a particular language, such as Mirage (OCaml) [64], HaLVM (Haskell) [89], or IncludeOS (C++) [18]. Applications targeting such unikernels have a lower porting time as the base libraries and/or language runtime have already been ported and optimized. On the other hand, these unikernels are less specialized than hand-crafted unikernels and therefore offer a lower degree of performance. Finally, unikernels that offer binary compatibility, such as OS$^V$ [52] or HermiTux [73]. These unikernels target apps that would normally run on Linux. They are characterized by a low porting overhead. Granted that all dynamically loaded libraries required by the application binary are satisfied in the unikernel, the porting time is practically zero. However, these unikernels present less potential for per-component specialization, and the binary compatibility interface is associated with additional costs. Furthermore, the binary compatibility interface has to be maintained in phase with the original kernel (Linux in the case of OS$^V$ and HermiTux) which might offset to some extent the low porting costs.

*processes* [28]. Having the OS personality running as part of the process offers a good potential for specialization. However, unlike unikernels, picoprocesses do not target single-purpose appliances and do not provide image size reduction. Furthermore, their specialization potential is limited to the OS personality, making top-down specialization up to resource management and scheduling impossible.

**Applications of Unikernels**

Note that not all unikernels target the cloud industry. Madhavapeddy et al. [65] show that unikernels can enable technologies such as just-in-time instanciation on embedded systems. More recently, De Simone and Mazzeo [25] highlight the potential of unikernels as an alternative to containers in embedded and real-time systems. Finally, other studies such as Lankes et al. [58] show applications of unikernels on bare metal in the context of high performance computing.

### 2.1.2 Characteristics and Architecture of Unikraft

Unikraft is an open-source unikernel framework that aims at automating the build of unikernels as much as possible, while providing a high specialization potential. The framework approach comes with the goal of facilitating building, optimizing, and sharing code among unikernels, platforms, and use-cases [56].

Unikraft is composed of a build system and a library pool. Operating system and application components are developed as pluggable libraries that can be configured and assembled by the build system. This approach can be summarized as *everything is a library*, a radical interpretation of library operating systems [56].

Unikraft's build system recognizes three types of libraries: (1) *main* libraries, (2) *architecture* libraries, and (3) *platform* libraries. Main libraries are very generic. They can define and implement decomposed operating system functionalities such as schedulers, memory allocators, or filesystems, but also user-space libraries (e.g., newlib) or applications (e.g., nginx). Note that Unikraft does not differentiate between application and library. From the perspective of the build system, an application is a library which provides a `main()` function. Architecture libraries such as `arm` or `x86` provide architecture specific primitives (e.g., atomic operations). Finally, platform libraries, such as `kvm`, `xen`, or `raspi` (Raspberry Pi platform) implement platform-specific services and functionalities. Note that platforms can be virtualized or not. Among others, the platform library is responsible for supplying the bootloader's entry point and device drivers (e.g., virtio), along with linker scripts that define the executable format (e.g., ELF for Xen, Multiboot for KVM).

Libraries can be *internal* or *external*. Internal and external libraries are technically very similar. The main difference is that internal libraries live within the Unikraft official repository[2] (*main tree*) and do not have external dependencies (i.e., the main tree is self-contained). *main* and *platform* libraries can be internal or external. *architecture* libraries, however, can only be internal.

Internal libraries define and implement the core Unikraft API, which is designed to resemble POSIX.1 [9]. Within the main tree, API definition is conven-

[2]`https://xenbits.xen.org/gitweb/?p=unikraft/unikraft.git`

```
./                              unikraft/
├── unikraft/                   ├── arch/
│   └── {}                      │   ├── x86/
├── apps/                       │   └── ...
│   ├── app-helloworld/         ├── lib/
│   └── ...                     │   ├── ukalloc/
├── libs/                       │   ├── ukallocbbuddy/
│   ├── lib-newlib/             │   └── ...
│   └── ...                     ├── plat/
└── plats/                      │   ├── kvm/
    ├── plat-solo5/             │   └── ...
    └── ...                     └── ...
```

Figure 2.2: Simplified Unikraft development tree. The main tree (`unikraft/`) hosts *main* (`lib/`), *platform* (`plat/`), and *architecture* (`arch/`) libraries. External main libraries live under `libs/` and `apps/`, external platforms under `plats/`.

tionally separated from implementation. The kernel API is defined by a specific set of internal libraries that provide interface headers, generic functionalities, and helpers. `uksched`, for instance, defines the scheduler API, and `ukalloc` defines the allocation API. APIs can be implemented by an arbitrary number of internal and external libraries. The main tree offers at least one implementation of each API element. `ukschedcoop`, for example, provides a cooperative implementation of the scheduler API, and `ukallocbbuddy` implements the memory allocation API as a buddy system. Figure 2.2 depicts a simplified Unikraft development tree, summarizing the different types of libraries within and outside of the main tree.

### Build Process

Functional unikernels can be assembled by the build system from a limited set of core components[3]:

- An architecture library (e.g., `x86`), which provides architecture abstraction for use by platform and main libraries.
- A platform library (e.g., `kvm`), which defines the image format and performs platform- and architecture-specific boot-time initialization.

[3]This list is slightly simplified. In practice, a small number of additional abstractions are required, but reviewing them outreaches the scope of this thesis. An exhaustive list is available in Section 5.3 as part of our evaluation of internal boot time.

- The `ukboot` main library, which performs generic kernel initialization.
- The `ukalloc` interface, which provides the internal memory allocation interface, and one implementation of the interface, such as `ukallocbbuddy`.
- A C standard library. The main tree provides a minimalist standard library called *nolibc*. nolibc is not a fully-featured libc but only provides the functionalities needed by internal libraries. newlib [4] and musl [3], two additional standard libraries, have been ported outside the main tree.
- Any library which provides a `main()` function (e.g., `helloworld`).

Libraries can be selected and configured via `make menuconfig`, which relies on a KConfig system similar to the Linux kernel [55]. Each library provides information to the KConfig system via a `Config.uk` file: configuration options, help messages, etc. Upon execution of `make menuconfig`, the KConfig system retrieves this information from the various libraries and fills the configuration menu. Architecture libraries, for instance, allow the user to specialize the image for a specific type of processor (via `-march` compiler options). Nginx allows individual modules to be enabled or disabled (`dav`, `scgi`, etc.). Figure 2.3 summarizes the assembly process of Unikraft images, from the perspective of the user.

### Porting Process

Each library directory contains one `Config.uk` file which contains information for the KConfig system, one `Makefile.uk` which contains build information, patches, and glue code. Porting means filling in `Makefile.uk` with paths to headers that need to be included, source files that need to be compiled, and compilation flags. Some applications also generate data or code at compile time; this has to be handled as well. Porting also means adapting the library to Unikraft's interface. This is facilitated by the fact that Unikraft offers an API very similar to POSIX. However, some libraries like C standard libraries require more changes. For instance, system call wrappers have to be modified to not issue a `syscall` instruction and instead call the appropriate system call procedure in Unikraft. Finally, Unikraft remains at an early development stage, and porting therefore also means implementing missing OS functionalities (e.g., missing system calls).

### Tracepoints

Unikraft supports *tracepoints* [7]. If enabled in ukdebug, data associated with the content of the timestamp counter (TSC) can be saved in an internal, fixed-size buffer. Tracepoints data comprises the tracepoint function name (*tracepoint*

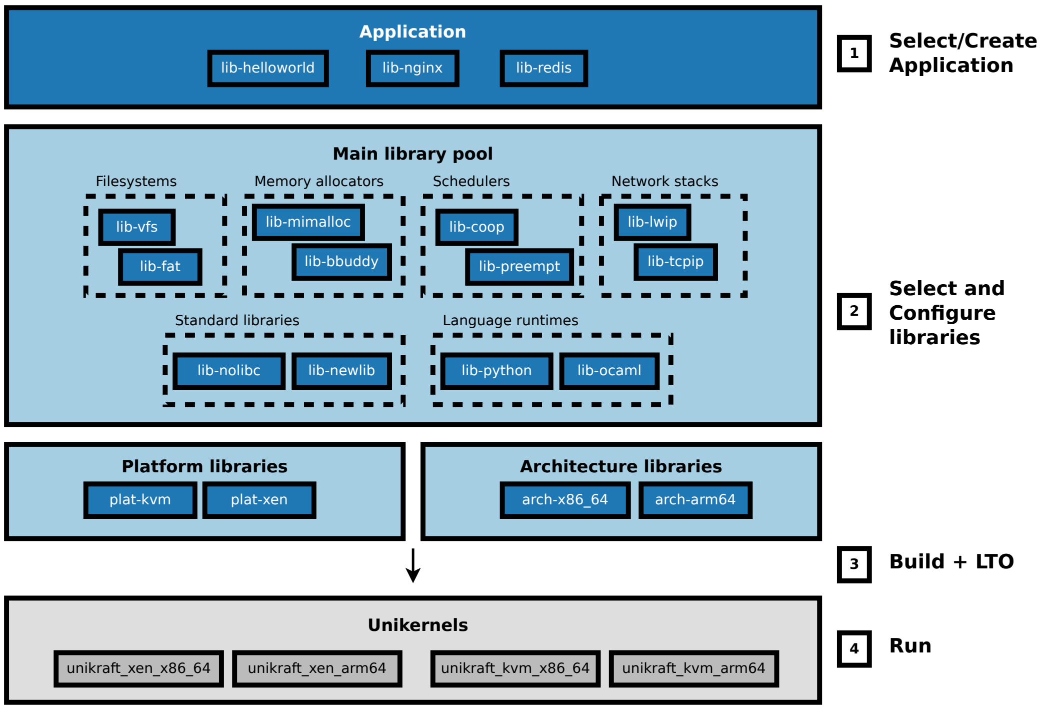


Figure 2.3: Unikernel building process in Unikraft, from the perspective of the user [56]. Users first select and configure a target application, along with necessary *main*, *platform*, and *architecture* libraries. Target unikernels are obtained after building, linking, and optimizing. The image format is defined by the platform library's linked scripts, which are executed at the third stage.

*label*) and up to 7 optional tracepoint parameters of any type. At each tracepoint write, the data is written to the buffer along with additional metadata. When the end of the buffer is reached, ukdebug automatically disables tracepoints. The content of the buffer can be retrieved externally at kernel shutdown or dumped to standard output. A set of `gdb` scripts is provided by the build system. Tracepoints are associated with a certain runtime overhead: disable and re-enable interrupts, multiple memory writes, pointer arithmetic, and a TSC read. Note that disabled tracepoints are automatically optimized out at build time.

### Conclusion

Compared to other unikernels such as Hermitux, OS$^V$, or Mirage, Unikraft's framework approach enables a greater potential for specialization and increased versatility. In particular, we expect that Unikraft's framework approach makes

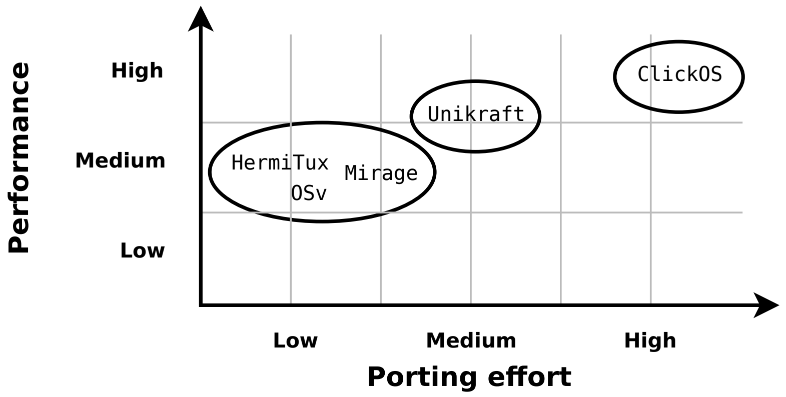


Figure 2.4: Position of the Unikraft framework on a performance v.s. porting effort plane compared to HermiTux, OS$^V$, Mirage, and ClickOS. Porting applications to Unikraft requires more effort but a higher degree of performance can be achieved due to a higher degree of specialization.

it easier and more elegant to implement allocator specialization. On the other hand, it suffers from increased porting costs and is significantly less mature than other older unikernel projects. Figure 2.4 depicts the position of Unikraft on a performance/porting effort plane, compared to other unikernel projects.

## 2.2 Dynamic Memory Allocation

The virtual address space of processes in general-purpose operating systems is typically organized in several sections called *segments*: the *text* segment, the *initialized* and *uninitialized* (bss) data segments, the *stack*, and the *heap* [50]. Figure 2.5 depicts a typical, non-randomized memory layout in Linux x86-32. The text segment contains program code. The initialized and uninitialized segments contain initialized and uninitialized global and static variables, respectively. Finally, the stack and the heap can be used by programs to allocate memory at runtime. The stack is a last-in-first-out (LIFO) data structure. When a function is entered, a new stack frame is created, containing function arguments and local variables. All variables are automatically freed when the stack frame is removed at function exit [50]. Memory allocated from the stack is therefore managed automatically. The heap takes a special role as its memory is explicitly managed by the program. For this purpose, the OS supplies the program a section of memory that the program is free to use as it sees fit. However, in practice, efficiently managing this memory is a non-trivial and error-prone task. For this reason, management of heap memory is typically the responsibility of a *dynamic*

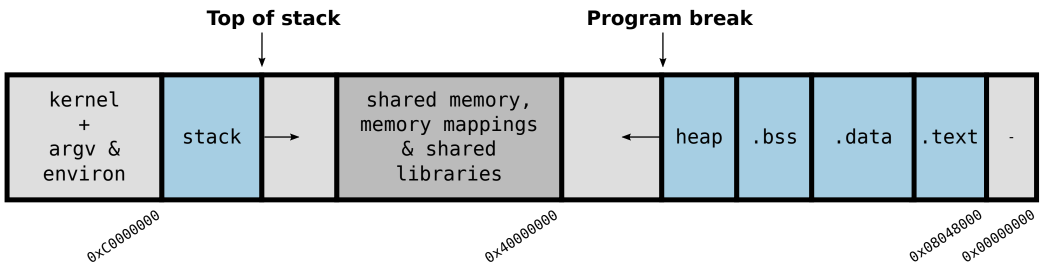


Figure 2.5: Typical memory layout of a process on Linux/x86-32, without Address Space Layout Randomization (ASLR) [50]. The stack grows downwards, the heap upwards. The upper limit of the heap is called *program break*. Memory mappings are done in between. The kernel is mapped at the top, after `0xC0000000`. With ASLR, all components would still be present but randomly placed across the heap effectively making the exploitation of memory corruption bugs harder [47].

*memory allocator* that manages the heap memory on behalf of the application.

Applications request heap space of a certain size through an allocation interface. If the request can be satisfied, the allocator responds with the address of a sufficiently large block of memory within the heap area, which is then used by the application independently of the allocator. If the allocator does not have sufficient space to satisfy the request, it can ask the kernel to expand the heap. The application can give back ownership to the allocator when it does not need the memory anymore. Memory allocation algorithms are as such *online* algorithms [91], required to irrevocably satisfy requests in sequence with minimal delay.

There is a consensus that memory allocators have a crucial impact on overall performance [14, 30, 43]. Berger et al. [14] show that programs can run as much as 60% faster just by switching to a more appropriate memory allocator. For this reason, allocators historically focused on minimizing runtime overhead while keeping memory usage as low as possible. Further research highlighted other determining factors such as runtime guarantees for embedded and real-time systems [69], security for memory unsafe languages [12], scalability in the context of multithreaded applications [13], or compiled size for cloud images. The memory allocation problem is therefore a complex issue, involving a variety of trade-offs. In fact, even the single problem of minimizing runtime and memory usage cannot be solved for all allocation patterns by a single allocator [91].

Nevertheless, it was shown that allocation patterns typically exhibit regularities within and across programs [91, 96]. If dynamic memory allocators cannot behave optimally in all cases, they only have to do so for realistic patterns. Two approaches to dynamic memory allocation are present in the literature: *general-purpose* allocation and *special-purpose* allocation (also referred to as *cus-*

*tom*). While general-purpose allocators attempt to provide good overall behavior for the most common allocation patterns, custom memory allocators ambition to offer optimal performance for a specific application or allocation pattern.

In this section, we first provide an overview of memory management in general-purpose operating systems. Then, we introduce general- and special-purpose memory allocation in further detail. Finally, we present the four memory allocators that will be studied in the course of this thesis.

## 2.2.1 Memory Management in General-Purpose OSs

This subsection offers an overview of memory allocation in general-purpose operating systems. We present the characteristics of memory allocation in user and kernel-space, and how they interact with each other.

### User-Space Memory Management

User space memory allocators manage heap memory. The size of the heap is determined by the position of the program break. The heap can be grown or shrunk at the allocator's request. Growing or shrinking the heap requires modifications in the page table, as this will create or remove memory mappings in the process' virtual address space. These operations can therefore only be executed by the kernel. In general-purpose operating systems, user- and kernel-space communicate via system calls. Following system calls are typically used in the context of dynamic memory allocation:

- The `brk()` system call moves the program break to the passed pointer and returns its previous position [50]. The success of the operation is conditioned by the validity of the passed pointer and the resource limits of the process. The `sbrk()` variant increments the program break by a passed integer.

- The `mmap()` system call creates new memory mappings [50]. The type of memory mapping is defined by the *flags* argument. `mmap()` can be used, among others, to map files in memory or map a new buffer of physical memory in the address space. Mappings can be modified using `mremap()` or removed using `munmap()`.

- The `madvise()` system call can be used by the program to provide additional information (*advice*) to the kernel about how it is going to make use of memory [50]. The advice can be that the program will read memory sequentially or randomly so that the kernel can adapt its read-ahead policy, or that the program does not need certain pages anymore.

- The `mprotect()` system call can be used by the application to ask the kernel to change the virtual memory protection of certain pages [50]. The application can request pages to be any combination of read, write, and execute[4].

Memory allocators should be conservative in their resort to system calls, as these are typically bound to high costs [84]. In particular, system calls that modify the size of the heap should be taken with care, as they trigger not only changes in the page table, but also force zeroing of newly allocated pages. Yang et al. [94] show that zeroing can be very expensive: up to 12.7% of all application cycles.

### Kernel-Space Memory Management

Kernel stacks are typically small and of fixed size [21]. For this reason, kernel components widely rely on dynamic memory allocation. On the one side, the kernel might simply require an arbitrarily sized contiguous buffer of virtual memory. This type of allocation is similar to user-space memory allocation and can be served by practically any memory allocator operating on the kernel virtual address space. On the other side, the kernel might require an arbitrarily sized buffer of virtual memory that maps to a contiguous buffer of *physical memory* [75]. This might be necessary when allocating buffers for direct memory access (DMA): external peripherals will access this memory without going through the memory management unit (MMU)[5]. Such allocations are typically handled by a specialized memory allocator that runs on a portion of memory that is always resident, mapped one-to-one with physical memory, and, if relevant, usable for DMA. In the Linux kernel, for instance, memory that is not guaranteed to be contiguous can be allocated via the `vmalloc` interface, and contiguous memory via the `kmalloc` interface[6] [39].

There is no consensus on whether kernel allocations should be contiguous in the general case (i.e., when there is no explicit requirement for contiguous memory). For instance, the recommended allocation interface in the Linux kernel is `kmalloc` [23], which provides physically contiguous memory. On the other hand, the FreeBSD kernel recommends the `malloc` interface [54], which does not guarantee contiguous memory. The main advantage of using contiguous memory in the kernel is that such allocations do not require modification of the page table

[4]Not all combinations are supported by all platforms.

[5]Alternative solutions such as input-output memory management units (IOMMU) or Scatter/Gather DMA do exist, but they are bound to additional costs, see Park et al. [75].

[6]Other allocation mechanisms exist in the Linux kernel, however, enumerating them would be beyond the scope of this work. An exhaustive list is be accessible as part of the Linux Memory Management Documentation: `https://www.kernel.org/doc/html/latest/vm/index.html`.

since there is already a one-to-one mapping. This results in a lower allocation overhead for the kernel. Second, these allocations can be registered as a single large page, resulting in a single TLB entry and reduced TLB misses [21]. The main drawback of a wide usage of contiguous memory is external fragmentation. A high degree of fragmentation could make finding large buffers of contiguous physical memory particularly difficult in situations of memory pressure, leading to decreased performance.

Kernel memory allocation interfaces are typically similar to POSIX with the addition of a few kernel-specific features [23, 54]. Most importantly, kernel allocators allow callers to specify whether or not sleeping or performing I/O is allowed during allocation. This is essential for use in interrupt handlers. Other options, for instance, can force the kernel to retry the allocation as often as possible until it finds a free buffer.

### 2.2.2 General-Purpose Memory Allocation

General-purpose memory allocators aim to offer good performance for most allocation patterns. They implement the standard allocation interface and typically make decisions based exclusively on size. This is an important difference with custom allocators, which typically integrate additional semantic information from the application. In the C language, general-purpose allocators implement the `malloc()` and `free()` family of functions, in C++ `new` and `delete`, etc.

General-purpose allocators make decisions at allocation and deallocation time based on *policies* [91]. Policies define the behavior of the allocator: where to place the requested memory block in the heap, whether or not to reuse freed memory, waste space by reusing freed blocks of a larger size, or split and/or coalesce freed blocks. Policies in general-purpose allocators are developed to leverage regularities across programs. Such regularities can be that objects allocated together are likely to be freed together [91], or that there is a high degree of temporal locality in the size of memory requests [40]. For instance, a policy resulting from such observations could be to store objects of same size together.

Memory allocators implement policies with different *mechanisms* [91]. Mechanisms are technical solutions to implement policies. Allocators can, for example, maintain a single linked list of free blocks that is linearly traversed to satisfy allocation requests (*sequential fits*, notably *first fit*, and *best fit*), maintain several linked lists for distinct size classes (*segregated fits*), or maintain a bitmap of free blocks (*bitmapped fits*). Not only different policies, but also different mechanisms can result in very different properties with regard to runtime performance, memory usage, real-time guarantees, etc.

The memory allocation research has been very prolific over the last two decades and a large number of mechanisms and policies have been introduced[7]. Since there is no strong theory behind program behavior or memory allocation, memory allocators are typically evaluated experimentally on a set of benchmarks and real-world programs. Despite the ad-hoc nature of general-purpose memory allocators, significant progress can be observed since the 1990s, and general-purpose memory allocators now perform very well on average. Nevertheless, the weaknesses of general-purpose memory allocators remain tangible and the claims of their shortcomings very present [81]. A historical remedy to this problem was custom memory allocation.

### 2.2.3 Custom Memory Allocation

Custom memory allocators (CMAs) specialize on a particular allocation behavior. They are typically employed as an optimization of general-purpose allocators, promising a gain in runtime performance, reduced memory consumption, or eased manual memory management [15]. Approaches to custom memory allocation differ in the level of coupling between application and allocator: some CMAs implement a specific, non-POSIX compliant API that requires direct support from application code, whereas others achieve specialization by profiling the application and synthesizing a custom allocator backend. This subsection summarizes the different approaches, and concludes the strengths, weaknesses, and trade-offs of custom memory allocation, and how they could be leveraged in the context of unikernels.

#### Manual Approaches

Some approaches rely on a tight coupling, such as *regions* [36] or *object stacks* [91], two historical approaches to special-purpose memory allocation. Regions allocate memory by incrementing a pointer within a chunk of memory. Freeing individual objects is not supported, so that all objects within a chunk have to be freed at the same time. Object stacks are very similar to regions. However, they implement a stack-like deallocation behavior: freeing an object automatically frees all objects allocated more recently in the stack. Both regions and object stacks rely on simplified bookkeeping to improve performance but require direct support from application code.

More recent approaches to tightly coupled memory allocation include hint-based allocation, such as DEFERO [45] (2007). DEFERO performs allocations based on size and *location*. The allocation function is given a size and a location hint,

[7]We present four allocators that rely on different policies and mechanisms in Subsection 2.2.4.

and tries to return a block which is the closest possible to the passed hint. Well chosen hints result in increased locality of reference and thus less TLB and cache misses. Other hint-based allocators include TP and Medius [46].

There is a consensus in the literature that tightly coupled CMAs such as regions, obstacks, and hint-based allocators can result in high performance increase. Berger et al. [15] (2002) find that regions can deliver performance improvements of up to 44% in runtime performance compared to the Lea allocator, a state-of-the-art general-purpose allocator at the time of their study. However, tightly coupled CMAs require a tedious and error-prone manual source code integration, greatly limiting their practicability in practice [46, 91].

### Per-Class & Container-Centric Approaches

Previous observation motivated the development of strategies that do not require manual modification of the source code. A historical approach is *per-class* allocation. Per-class allocators implement the standard allocation interface but specialize on a single object type or size. In C++, such allocators can be associated to a specific class by overloading the `new` and `delete` operators [14]. Per-class allocators can be easily written using Berger et al. [14]'s heap layers, an allocator framework based on C++ templates that allows fast implementation of custom memory allocators from reusable parts.

Note that there is little consensus about the performance impact of per-class allocators. Berger et al. [15] (2002) explain that, while per-class allocators did certainly perform well in the past, modern general-purpose allocators perform very well in situations where many objects of a same size are allocated, making per-class allocators essentially superfluous.

A more recent approach called *container-centric* allocation combines C++ STL containers[8] and hint-based CMAs like DEFERO, TP or Medius [46]. Container-centric allocation leverages the fact that STL containers allocate their elements independently. By modifying the STL library, containers can be instrumented to automatically pass a hint to the underlying allocator, maximizing for example locality of reference within a container. Performance increase can therefore be achieved without introducing additional complexity in the application.

Container-centric approaches have shown to produce very good results at low cost. Jula and Rauchwerger [46] show that container-driven allocation produce runtime improvements of 7% and 17% on average, compared to the Lea and FreeBSD allocators, respectively. However, container-centric approaches have a limited scope. DEFERO, TP, and Medius in particular can only be considered in C++ applications that make use of STL containers.

[8]We are here referring to holder objects from the C++ standard library, such as `std::vector` or `std::list`, not lightweight virtualization.

### Profile-guided Approaches

Finally, certain strategies rely on loosely coupled memory allocators that leverage program-specific heuristics through the standard allocation interface. An early example of a loosely coupled custom memory allocator is Grunwald and Zorn's CustomAlloc (1993) [40], a system which automatically synthetizes custom allocators from previously recorded allocation traces. Programmers write their application using the standard POSIX allocation interface. Then, CustomAlloc runs the application a number of times, analyzing the allocation behavior for different inputs. Following the analysis, it synthetizes a CMA that maintains distinct free-lists for the most common allocation sizes. Free-lists are managed differently by a *fast* or a *general* allocator depending on statistical data from the analysis. The fast allocator trades memory usage for performance, and statistical data help to find an optimal trade-off between memory usage and performance.

The CustomAlloc approach shows that there is a potential in generating memory allocators from allocation traces. Nevertheless, such approaches take the risk of being mislead by trace data [88], and suffer from limited portability with hard-coded customization at compile-time [17, 61].

Savage and Jones [81]'s halo (2020), a very recent allocator takes a similar approach. halo profiles the application and applies grouping and identification algorithms to determine how allocated data is used. At runtime, it uses previously gained clustering information to maximize temporal locality, adjacently placing data that is used at the same time. In this manner, halo demonstrates speedups up to 28% over jemalloc [81], at the cost of build-time analysis and increased memory usage.

### Conclusion

We conclude that manual approaches to special-purpose memory allocation can produce good results with regions or hint-based allocators. However, manual approaches are not practicable in many cases due to their tedious and error prone porting process. Less coupled approaches such as container-centric allocation can also produce good results but they are restricted to specific applications that make heavy use, for example, of C++ STL containers. Finally, profile guided approaches such as halo can have very good results but at the cost of memory usage and compile time.

We see a clear benefit in applying loosely coupled CMA technologies such as halo, Defero, tp, or Medius but only in cases where it fits with the application, and the trade-offs (e.g., more memory usage for more performance) benefit the user. The need for flexibility thus advocates the use of allocator specialization in unikernels.

### 2.2.4 Memory Allocators in Unikraft

This subsection introduces Unikraft's buddy system (which we will refer to as BBUDDY), Mimalloc [62], TLSF [69] and tinyalloc [82]. These allocators will be studied in Unikraft as part of Section 4.2.

#### BBUDDY

BBUDDY is Unikraft's binary buddy allocator. It is based on MiniOS's `mm.c` page allocator[9], an implementation of Knuth's buddy system [53]. BBUDDY is a page allocator and as such it only allows for allocation and deallocation at page granularity (4 KiB on x86, huge pages are not supported by Unikraft as of 0.4.0). A set of external wrappers called `ifpages` (page allocation interface) implements the standard POSIX allocation interface on top of BBUDDY's page allocation primitives. In particular, allocations under a page are padded to a whole page, potentially leading to significant internal fragmentation. BBUDDY supports non-contiguous memory, called *memory regions* in Unikraft. These might happen when the Unikraft kernel is booted with an initial ramdisk image.

BBUDDY is a classical buddy system, however, its initialization function differs slightly from the original algorithm. The initialization function is passed a linear memory block of arbitrary size. After reserving space for metadata, it splits the linear block into smaller blocks of size $2^{n+s}$ bytes with a so-called *order* of $n$ ($n > 0$). The page shift is defined as $s := log_2(\texttt{page\_size})$ and guarantees that blocks are not smaller than the page size. $n$ is constrained by the amount of memory passed to the initialization function. In addition to this, blocks need to be contiguous and aligned to their own size. Alignment constraints are leveraged to ease block coalescing. Note that, as a consequence of previously mentioned constraints, the initial linear block of memory does not need to be a power of two. BBUDDY exposes an `addmem` function which allows to repeat this process for additional memory regions.

There are $\texttt{sizeof(void*)} \cdot 8 - s$ free-lists (i.e., 52 for 64 bit systems and a page size of 4 KiB). Lists of order $n$ point to free blocks of size $2^{n+s}$ bytes.

The allocation function goes through the free-lists in increasing order starting at the requested size, taking the smallest block order that can be used to satisfy the request. The chunk is unlinked from the free-list and halved as often as necessary to obtain a block of sufficient order. The halves (also called *buddies*) are added to the corresponding free-lists.

The deallocation function finds the region corresponding to the supplied pointer. In order to speedup the identification of mergeable buddies, BBUDDY maintains a page bitmap that marks each page as allocated or free.

[9] `https://xenbits.xen.org/gitweb/?p=mini-os.git;a=blob;f=mm.c`

Besides the free-lists and the page bitmap, BBUDDY maintains a free page counter for statistical purposes. Figure 2.6 summarizes BBUDDY's metadata and heap layout.

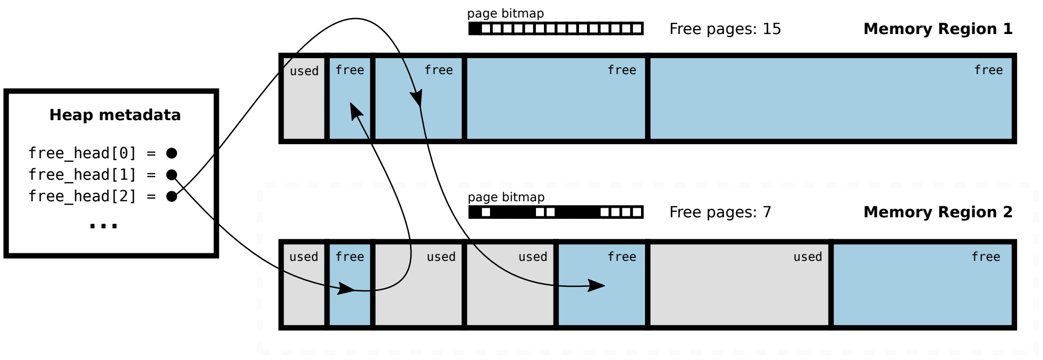


Figure 2.6: Heap metadata in BBUDDY. Allocator-wide segregated free-lists reference free chunks of a given order, across regions. Page bitmaps allow for fast page status verification (used or free) during buddy coalescing. A free page counter allows for fast lookup of per region memory usage for statistics purposes.

### Mimalloc

Mimalloc was introduced by Leijen et al. [62] (2019) as part of their work on the functional languages Koka and Lean at Microsoft Research. Mimalloc features a good support for multithreaded workloads, overall high performance, a small code base and has a secure variant called *smimalloc*.

Mimalloc is centered around a *fast* and a *slow* path. The fast, highly optimized malloc and free paths are used most of the time. The fast allocation path satisfies requests with minimal bookkeeping, executing a single conditional branch and pop operation. The slow path, guaranteed to be taken regularly, performs housekeeping operations.

A fundamental observation behind Mimalloc's design is that memory allocators are now frequently used as back-end implementation of languages such as Python. These languages often leverage reference counting to free memory, typically leading to a large number of free calls when the execution exits a certain code block. This sudden, irregular, and large amount of free calls is undesirable as it introduces 'pauses' within program execution. Mimalloc addresses this via a so-called *deferred-free* mechanism: applications can define a limit of free calls to execute in a row. After hitting this limit they can pass remaining variables to a deferred-free-list which will be processed as part of the slow allocation path.

Mimalloc's good support for multithreaded workloads is achieved via a *lock-free* design. Each thread has its own heap. Heaps are organized in *segments* that contain so-called *pages*, themselves containing objects of a given size class. Size classes are multiples of the machine word size (8 bytes on 64-bit systems), up to 8 KiB. Over 8 KiB, segments only contain a single page and are always allocated from the slow path. Note that these pages are not OS pages; they are closer to what is called *superblock* or *subslab* in other allocators. The central articulation of Mimalloc's lock-free design is the concept of *sharded free-lists*. There are three free-lists per page. The first one is the standard free-list, used as a source of free blocks by the fast path. The second one, called the *thread-free* list, contains blocks freed by other threads updated in a lock-free manner via atomic push operations. The third one, the *local-free* list, contains blocks freed by the local thread. Both the local- and thread-free-lists are merged into the standard free-list as part of the slow path. The local- and thread-free-lists are therefore keys to a regular journey in the slow path: whenever the fast path encounters an empty standard free-list, the slow path is triggered to merge the local- and thread-free-lists or request more memory from the OS.

It is no accident that free-lists are present per page and not per segment; this extreme free-list sharding increases locality of reference. The authors indeed observe that applications tend to allocate structures containing elements of the same size class. In the case where there is only one free-list per size class, elements are likely to spread all across the heap, or at least all across pages of a size class. Having one free-list per page forces elements to be on the same page as far as possible. Figure 2.7 summarizes Mimalloc's heap layout.

#### TLSF

The Two-Level Segregated Fit (TLSF) memory allocator was introduced by Masmano et al. [69] to address the lack of good memory allocators in the context of real-time operating systems. TLSF features allocation and deallocation with a temporal cost of $O(1)$, along with a small and bounded fragmentation behavior. These characteristics contrast with most common allocators which typically feature unbounded worst-case temporal cost and fragmentation.

TLSF focuses on small systems where memory is a scarce resource, possibly without memory management unit (hence without virtual memory). The memory model is especially adapted for linear address spaces. There are no security features; the environment is considered trusted, and protection is done at user interface level.

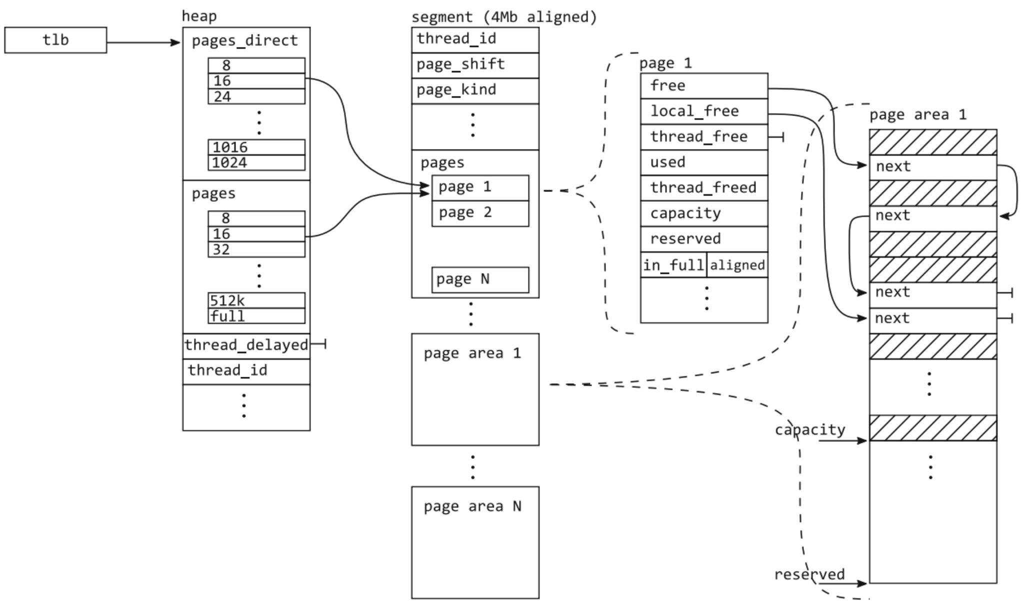


Figure 2.7: Mimalloc per-thread heap layout [62]. We observe the presence of segments holding pages of a same size class. Page metadata and page areas are separated by a guard page. There are three free lists per page, a consequence of the extreme *free-list sharding* policy.

TLSF relies on a *good fit* strategy, which can be considered the bounded counterpart of *best fit*, a strategy known to produce the least fragmentation. The implementation is based on segregated lists: a number of independent and ordered free-lists provide blocks whose size is between one size class and the following one. Good fit and segregated lists are used consistently for all size classes.

Typical segregated fits have a single list holding pointers to per size-class free-lists. The authors highlight that searching these lists can be costly if their number is high, which is precisely TLSF's case. They solve this problem by using two-level segregated fits, another level of indirection. TLSF's first level free-list splits in coarse power of two size classes. The second level free-list splits power of two size classes linearly in SLI size classes, where SLI is a user-supplied parameter. The first and second level free-lists are associated with a bitmap that allows fast lookup whether a segregated list still holds free blocks or not. Note that this search can be done in constant time due to hardware support for bitmap search (find first set operations, e.g., `lzcnt` instruction on x86). As a result, `malloc()` can be performed with an asymptotic behavior of $O(1)$.

TLSF splits blocks when necessary and follows an immediate coalescing policy: while this might not be the most efficient strategy, it reduces unpredictability and fragmentation. Memory blocks are only split above a threshold of 16 bytes: Masmano and others observe that the majority of allocations are not simple data such as pointers and integers, but larger structures. Having such a limit allows to store metadata within free blocks, which limits memory overhead.

Block headers contain size information, along with a pointer to the beginning of the previous block. This pointer is used for coalescing purposes upon inserting a newly freed block to the free-lists. Free blocks contain pointers to the previous and next free blocks as well, hence the requirement for a split limit of 16 bytes. Figure 2.8 summarizes TLSF's two-level segregated fit mechanism.

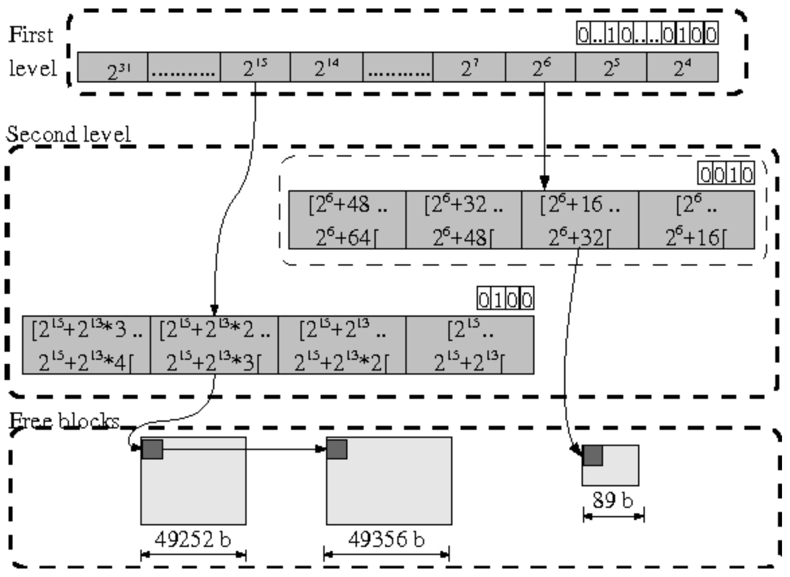


Figure 2.8: TLSF's Two-Level Segregated Fit mechanism [69]. The first level free-list separates in size classes that are a power of two apart. The second level splits power of two size classes linearly in user-supplied SLI size classes. The first and second level free-lists are associated a bitmap for fast free block lookup, allowing block search in $O(1)$.

### tinyalloc

tinyalloc [82] is a minimalist implementation of `malloc()` and `free()`. It is meant for use in systems with unmanaged linear memory such as WebAssembly or embedded systems.

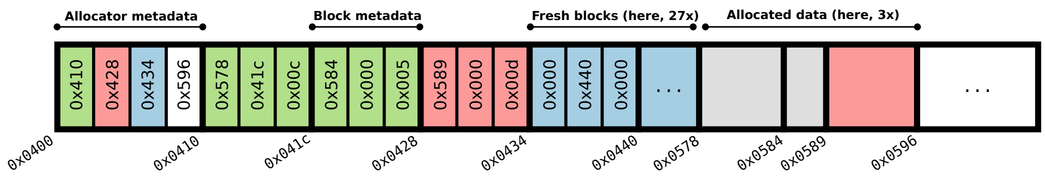


Figure 2.9: tinyalloc heap layout [82]. Allocator metadata contain a pointer to the first free block (in green), a pointer to the first used block (in red), a pointer to the first fresh block (in blue), and a pointer to the heap top (in white). Block metadata contain the address of the associated heap chunk, a pointer to the next block in the list, and the size of the block. In this diagram, there are two freed blocks (1x12 bytes, 1x5 bytes), one used block (1x13 bytes), and 27x remaining fresh blocks (the allocator was therefore initialized with 30 fresh blocks).

tinyalloc keeps three lists of blocks. The first one holds so-called *fresh blocks*. Fresh blocks are blocks that are uninitialized and not associated with actual memory. Fresh blocks are allocated at initialization time and only available in a fixed, user-configurable number. The second list holds *used blocks*. Used blocks are associated to a chunk of heap memory and are currently in use by the application. The third list holds *free blocks*. Free blocks are used blocks that have been freed. Unlike fresh blocks, free blocks have already been initialized and associated to a block of memory.

Upon receiving an allocation request, tinyalloc applies a *first fit* (sequential fit) policy, scanning through the entire free-list for a sufficiently large block of memory. Block splitting is optional, with a configurable split threshold. If no sufficiently large block is found, tinyalloc reserves space on the heap by increasing the heap pointer. A fresh block is then initialized with the newly allocated memory and added to the used list. Note that this is only possible if there are remaining fresh blocks. Otherwise, the allocation simply fails. The number of fresh block therefore has to be chosen carefully.

The heap is split in two parts: the one currently in use, split in chunk represented by free and used blocks, and the rest waiting to be associated to fresh blocks. The separation is done via a *top pointer* that points to the first byte of unassociated memory. Figure 2.9 summarizes tinyalloc's heap layout.

# Chapter 3

# Analysis

The previous chapter introduced Unikraft and how it achieves performance increase by fully specializing operating system components toward the application. We introduced the memory allocation problem and emphasized that dynamic memory allocators hold untapped specialization potential that can be leveraged to optimize for specific key performance indicators (KPIs). We observed the need for a solution that takes advantage of the particularities of applications without paying the price of manual specialization. This motivates us to study how memory allocators fit in Unikraft's *everything is a library* specialization strategy, whether choosing from a range of general-purpose allocators can be considered a trade-off between custom- and general-purpose allocation, and how this specialization affects KPIs.

In this chapter we further study general-purpose memory allocation in the context of unikernels. Section 3.1 discusses the differences between unikernels and general-purpose operating systems with regard to memory allocation. We are interested to determine how these differences affect the design of allocators in unikernels, and therefore how they impact allocator specialization strategies. Section 3.2 analyzes how to best leverage memory allocators as a specialization layer in unikernels and proposes a concrete specialization experiment that is based on the Unikraft framework.

## 3.1 Dynamic Memory Allocation in Unikernels

The vast majority of memory allocation research is realized in the context of general-purpose operating systems. To the best of our knowledge, no previous work in the literature specifically discusses memory allocation in the context of unikernels. However, we know that unikernels and general-purpose OSs present significant differences that are also relevant to dynamic memory allocation: unikernels run as a single process, in privileged mode, with a single address space, making expensive system calls as cheap as simple procedure calls [52]. We therefore see the need to examine if such differences in the execution environment can lead to new conclusions regarding memory management in unikernels and how such conclusions might drive allocator specialization strategies.

We have shown in 2.2.1 that general-purpose operating systems perform dynamic memory allocation in kernel and user space, and noted that both modes typically possess very different requirements. We are interested in better understanding whether these properties still exist in unikernels, and what they imply in the context of allocator specialization. In particular, we want to determine whether the separation of kernel-space and user-space allocators still makes sense in unikernels. Subsection 3.1.1 reviews the special aspects of kernel memory allocation in general-purpose OSs, and their relevance in unikernels. Subsection 3.1.2 analyzes how the characteristics of unikernels affect memory allocation at the application level.

### 3.1.1 Memory Allocation in the Kernel

We showed in 2.2.1 that kernel memory allocators need to (1) guarantee isolation between user and kernel space, (2) optionally allocate memory that will always be resident, (3) provide physically contiguous memory upon request, and (4) provide atomic behavior in critical execution paths. This subsection analyzes whether these requirements still apply to memory allocators in unikernels. Following this, we conclude on the possibility to share a memory allocator between kernel and application in unikernels.

**Isolation** The classical approach to unikernels considers the hypervisor as the trusted computing base. Unikernel-based VMs rely on the hypervisor to provide isolation, run entirely in privileged mode, and do not provide internal isolation. Hence, there is no concept of a user space, and as such no fundamental security reason why kernel and application should not share the same memory allocator.

Note that the absence of intra-unikernel isolation was criticized by recent work [74, 85]. In particular, Sung et al. [85] stress that while numerous unikernels are written in memory safe languages [59, 64], the use of unsafe primitives (e.g., inline assembly, dereferencing of raw pointers) is unavoidable in an OS-context. The authors argue that the absence of internal isolation between safe and unsafe unikernel components discards the benefits of using memory safe languages. Based on this observation, they show that isolation between memory safe kernel code, unsafe kernel code, and user space can be re-introduced at low cost on the basis of Intel Memory Protection Keys (MPK), ultimately entailing a separation of kernel and application memory allocators.

Thus, the necessity of separating memory allocators should be viewed in relation to the memory safety model of unikernels. While classical unikernels will draw no security benefits from such a separation, multiple allocators coupled with protection domains can back up memory safety in unikernels that rely on memory safe languages.

**Resident Memory** Many components of general-purpose kernels depend exclusively upon unpaged memory (i.e., memory that is always resident in memory). An essential reason for this is that page-fault handlers need to page in non-resident memory from disk, resulting in I/O operations. Thus, page-faults must not happen in atomic contexts. This observation is also valid for unikernels. Unikernels that implement paging (and in particular page eviction) therefore need to provide a kernel allocator that can supply unpaged memory.

Supporting paged kernel memory is a trade-off between the added complexity of critical sections and the potential gain in memory footprint. For this reason, paged kernel memory is not a widely supported functionality among general-purpose OSs. Linux and FreeBSD, for instance, do not support it [34, 39]. Unikernels are known to have a small memory footprint, and as such we expect that they will draw little benefit from supporting paged kernel memory.

Note that unikernels do not typically perform complex paging. IncludeOS [18], for example, operates with virtual memory switched off, that is directly on physical memory. Unikraft supports virtual memory but relies on a one-to-one physical to virtual memory mapping. In such cases, there is an implicit guarantee of memory to be resident. $\mathrm{OS}^{\mathrm{V}}$ [52] provides more complete support for virtual memory but does not support page eviction and demand paging can be disabled for kernel allocations.

**Physically Contiguous Memory** The requirement for physically contiguous memory also applies to unikernels. Many unikernels [18, 52, 56, 64], for instance, implement virtio paravirtualized drivers, which require host-guest communication channels (e.g., *virtqueues*) to be physically contiguous [86]. The requirement for physically contiguous memory is satisfied differently among unikernel projects. Unikraft addresses this problem by relying on a one-to-one virtual memory mapping, whereas IncludeOS [18] operates directly on physical memory. In this case, a single allocator can be shared between kernel and application. The virtual memory management in $\mathrm{OS}^{\mathrm{V}}$ instead supports explicit allocation of physically contiguous memory by incorporating knowledge of the state of the physical memory into the kernel allocator [52].

**Atomic Behavior** General-purpose kernels often implement complex mechanisms for memory release as part of the allocation path. When a situation of memory pressure is detected, the kernel can put the calling process to sleep in order to release some memory from caches or write buffers to disk [23, 54]. As a consequence, memory allocations can result in blocking behavior. Kernel memory allocators therefore take an argument that enables or disables blocking behavior, allowing the use of dynamic memory allocation in critical paths. This equally applies to unikernels. OS$^{V}$, for instance, implements the *Shrinker* API [52] which allows system components to register callbacks that will be called when the system is low on memory, possibly resulting in filesystem operations within the allocation path. OS$^{V}$'s kernel allocator thereby provides the caller the option to choose between atomic and non-atomic behavior. Unikraft, on the other hand, does not implement such a mechanism. Memory allocation is implicitly a non-blocking operation.

We conclude that kernel memory allocation in unikernels is not very different from kernel memory allocation in general-purpose operating systems. Unikernels also need dynamic memory allocators that can provide, upon request, physically contiguous memory, unpaged memory, or atomic behavior. The need for isolation is not present in the classical approach to unikernels, but recent work showed that there are meaningful applications for reintroducing a certain degree of isolation within unikernels.

We note that the requirements for memory contiguity, unpaged memory, or non-blocking behavior can be satisfied implicitly by the memory allocator in certain unikernels. Unikernels that do not implement paging (and more specifically, page eviction), operate on one-to-one mapped memory ranges, and do not block in the allocation path do not technically require different allocators for kernel and application[1]. This is, for example, the case in Unikraft and IncludeOS. Nevertheless, we emphasize that relying on a single allocator for kernel and application is not necessarily a good decision from the perspective of performance. Decades of experience in general-purpose operating systems showed that the kernel is a good candidate for custom memory allocation, and using specialized allocators such as object pools has the potential to drastically improve performance [17, 29, 39]. If technically possible, the decision whether or not to share an allocator between kernel and application should therefore be an informed decision based on KPIs (boot time, binary size, runtime performance, etc.) and the selected kernel feature set. Developing a kernel allocator for Unikraft is outside the scope of this thesis. However, Unikraft's ability to have several memory allo-

[1]This also applies to unikernels that do implement these functionalities but allow users to disable them for specialization purposes.

cation backends associated to different memory pools significantly facilitates the implementation of a kernel-specific allocator. For instance, a specialized kernel allocator could be assigned a one-to-one mapped memory range.

### 3.1.2 Memory Allocation in the Application

We recognize two characteristics of unikernels that are especially relevant in the context of memory allocation: (1) absence of system call overhead, and (2) no symmetric multiprocessing (although some unikernels do support it). While these characteristics do not fundamentally change the functioning of dynamic memory allocators — in fact, any user-space allocator should work just fine in a unikernel environment, we believe that some allocators might leverage them better than others.

#### System Call Overhead

We have shown in 2.2.1 that user-space memory allocators use system calls to grow or shrink the heap (`mmap()`, `brk()` and `sbrk()`), modify the protection of specific pages (`mprotect()`), or pass on memory usage hints to the operating system (`madvise()`). We showed that system calls have historically been associated with significant costs (we will reevaluate these costs in 3.1.3), and concluded that user-space memory allocators need to balance the cost of system calls with virtual memory usage, security, and performance. In the context of unikernels, however, this trade-off is very different, as the entire code runs in privileged mode. System calls are therefore simple procedure calls and applications can make heavy use of them to increase the coupling between kernel and application code. This has two fundamental consequences in the context of memory allocation. First, the allocator can grow and shrink the heap at minimal cost, and second, the allocator can modify page protections at minimal cost.

The ability to resize the heap at reduced cost can be leveraged in unikernels that rely on dynamically sized heaps like OS$^V$ [52] and Hermitux [58]. Even if unikernels host a single application (reducing the relevance of memory fairness), the application still shares the physical memory with the kernel. It is therefore important to avoid situations of kernel memory pressure caused exclusively by a waste of memory in the application. The other way around, the application should not be limited by an excessively conservative heap size. Note, however, that unikernels which rely on statically sized heaps like Unikraft or on a single kernel-user heap like IncludeOS [18] do not benefit from an increased ability to resize the heap at runtime.

System calls have been a historical limitation of attempts at providing memory safety [24, 26]. Therefore, the ability to modify page protections and communicate memory usage advice to the kernel code at low cost can be leveraged to mitigate the performance overhead of security-focused allocators, enabling, for instance, widespread reliance on guard pages.

#### Single-Core Support

Unikernels frequently assume a uniprocessor model. Unikraft, in particular, runs on a single core with cooperative scheduling. Therefore, no locking is required as long as threads do not yield within `malloc()` [33]. We observe that this might disadvantage allocators with support for multithreading that do not provide optimizations for uniprocessor systems. Such optimizations can be to avoid infrastructure for per-processor arenas or heaps [33], relax synchronization, or remove per-thread metadata [70]. Previous research showed that such optimizations can improve performance by as much as 15% [70].

### 3.1.3 Impact of System Calls

We have previously emphasized that the absence of classic system calls are a major difference between unikernels and general-purpose operating systems. We are interested to quantify this. In particular, we see the need to estimate the performance gain that can be offered by unikernels with regard to memory allocation and how this impacts the cost of temporal memory safety in unikernels. In this subsection, we first present an analysis of the cost of system calls in general-purpose operating systems and compare it to the cost of a procedure call. Then, we analyze the number of memory allocation related system calls performed by different general-purpose and security-focused allocators.

Several studies in the literature such as Soares and Stumm [84] (Linux kernel) and Rittinghaus [78] (Microsoft Windows) have already quantified the cost of a system call in different general-purpose operating systems. However, these measurements are about a decade old and we are not aware of any more recent work on this matter. The absence of recent measurements is problematic since the performance of core OS functionalities is known to have decreased significantly in the last few years [77]. Spectre and Meltdown mitigations, in particular, account for a significant part of this performance decrease, and kernel page table isolation (KPTI) [41] is already known to have a significant impact on the cost of system calls [22]. This cost, however, has not been precisely quantified in the literature. We therefore provide more recent measurements that take recent changes into account.

### Experimental Setup

Our measurements are run on a GNU/LINUX Debian Buster system with a Linux kernel version 5.6.6, built from source. The machine is equipped with 1x Intel® Xeon® E5-2690 v4 with 2.6Ghz base frequency and an invariant timestamp counter (TSC). Hyper-threading is disabled.

### Cost of System Calls

We want to benchmark the cost of a system call and compare it with the cost of a procedure call. The cost of a system call can be separated into (1) *direct* costs (mode switch), and (2) *indirect* costs (pipeline flushing, pollution of TLB, processor caches, etc.) [84]. Direct system call costs can be measured from user space via a *null* system call, which returns a constant value. By measuring the time elapsed between the execution of the `syscall` instruction and its return, we have a precise measurement of the mode switch cost. Indirect costs are significantly more complicated to measure. Indirect costs will be lower in unikernels (no `syscall` instruction, hence no pipeline flushing), but still exist as the execution of the function call will still alter the TLB and cache.

In this subsection we focus on mode switch costs, providing us with a lower bound estimate of the cost of system calls. We proceed as follows. First, we implement a null system call in a Linux 5.6.6 kernel. Then, we (1) determine the measurement overhead by doing two TSC reads sequentially and taking the difference, (2) measure the time using the TSC it takes to execute one million (non-inlined) function calls that return a constant value, and (3) benchmark the time it takes to perform one million null system calls. The experiment is repeated 100 times, the measurement overhead is subtracted, and average results are taken. Note that we leverage the `cpuid` instruction to ensure the serialization of the TSC reading (`rdtsc`).

The experiment is run three times, one with the default kernel configuration, one disabling kernel page table isolation (`nopti` kernel argument), and one disabling all recent security mitigations (`mitigations=off` kernel argument [2]). Table 3.1 shows our results. We observe that the cost of system calls has risen by x4.5 with recent CPU vulnerability mitigations (KPTI, in particular). While this has not been observed in the literature at the time of this writing, we find that this is consistent with Corbet [22] and Ren et al. [77]. The results without security mitigations are consistent with Soares and Stumm [84]. We observe that system calls are on average 111x more expensive than procedure calls (25x without security mitigations). These numbers are significant in the context of memory allocation; 665 CPU cycles is equivalent to more than 30 allocation calls with a modern general-purpose allocator, assuming a per-allocation cost of 20-

| | Security Mitigations | System Call | Function Call |
|---|---|---|---|
| GNU/LINUX 5.6.6 | default | 665 | 6 |
| | default + `nopti` | 202 | 6 |
| | `mitigations=off` | 149 | 6 |

Table 3.1: Cost of function and system calls in Linux 5.6.6, with the default configuration, without KPTI (`nopti`), and without security mitigations (`mitigations=off`, which includes `nopti` [2]), in CPU cycles.

25 cycles [48]. The cost of a procedure call does not vary between the different runs, which is to be expected since programs do not interact with the kernel as part of a procedure call.

### System Calls in Memory Allocation

We run *barnes* [11] and *espresso* [80] and record the number of system calls performed by the applications with varying memory allocators (smimalloc, mimalloc[2] [62], snmalloc [63], jemalloc [33], and hoard [13]). System calls are recorded via `strace`. barnes simulates the gravitational forces between 163840 particles. espresso is a programmable logic array analyzer. Both are very common programs in the allocation literature [13, 15, 62]. These applications are considered to represent a typical allocation pattern, which justifies our choice for this study. smimalloc, mimalloc, snmalloc, jemalloc, and hoard are well studied allocators that leverage different allocation strategies[3].

Tables 3.2 shows our results. We observe that the number of system calls performed by allocators varies significantly. In particular, smimalloc, the secure variant of mimalloc, performs notably more `mprotect()` system calls: 7x and 34x more than mimalloc for barnes and espresso, respectively. Note that smimalloc only offers weak security guarantees in comparison to state-of-the-art security-focused allocators such as *Oscar* [24]. In particular, smimalloc does not provide protection against use-after-free. We assume that Oscar, which performs one system call per allocated heap chunk, would present a significantly higher number of system calls. Unfortunately, we are not aware of any open source implementation of Oscar and were therefore unable to integrate it into this benchmark.

[2]See 2.2.4 for an introduction to mimalloc and smimalloc.

[3]Note that we did not include tinyalloc, TLSF, and BBUDDY as these allocators operate on a static chunk of memory and therefore do not perform any system calls.

| | all | `brk()` | `mmap()` | `munmap()` | `madvise()` | `mprotect()` |
|---|---|---|---|---|---|---|
| smimalloc | 177 | 4 | 38 | 2 | 0 | 90 |
| mimalloc | 100 | 4 | 38 | 2 | 0 | 13 |
| snmalloc | 111 | 4 | 45 | 2 | 0 | 15 |
| jemalloc | 141 | 5 | 56 | 6 | 1 | 17 |
| hoard | 155 | 4 | 62 | 23 | 0 | 17 |

(a) *barnes*

| | all | `brk()` | `mmap()` | `munmap()` | `madvise()` | `mprotect()` |
|---|---|---|---|---|---|---|
| smimalloc | 44699 | 4 | 35 | 2 | 0 | 438 |
| mimalloc | 44274 | 4 | 35 | 2 | 0 | 13 |
| snmalloc | 44285 | 4 | 42 | 2 | 0 | 15 |
| jemalloc | 44343 | 5 | 50 | 6 | 32 | 17 |
| hoard | 44336 | 4 | 69 | 20 | 0 | 17 |

(b) *espresso*

Table 3.2: Number of system calls performed with varying allocators and workloads. The *all* column displays the total number of system calls performed by the application, including system calls that are unrelated to memory management.

### Conclusion

We have shown that the cost of system calls has increased considerably in the last few years due to security features such as kernel page table isolation. We have recalled that current approaches to provide temporal memory safety suffer from an increased system call overhead and illustrated that this is also the case of hardened allocators such as smimalloc. The cost of such allocators has therefore intensified over the last few years and unikernels can leverage the absence of system call overhead to offer state-of-the-art temporal memory protection at lower cost.

## 3.2 Specializing Memory Allocators

Section 2.1 introduced the typical key performance indicators of unikernels: image size, boot time, runtime memory usage, runtime performance, and security. This section presents these KPIs in greater detail and analyzes how memory allocators can be leveraged to tune unikernel environments for them. We put our findings into practice by porting a selection of memory allocators to Unikraft: BBUDDY, tinyalloc, TLSF, and Mimalloc. This section motivates this choice and our expectations that these memory allocators will present optimal behavior in different contexts and KPIs. Figure 3.1 presents the interdependence between KPIs, graphically summarizing our work in this section.

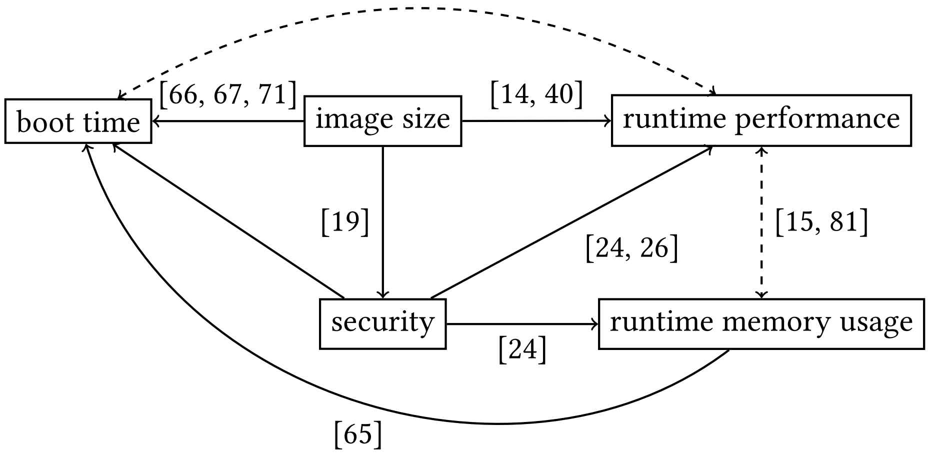


Figure 3.1: Graphical representation of the correlation between unikernel KPIs. A full arrow represent an *influences* relation. Dashed arrows represent a trade-off situation where one KPI can be traded off for the other and vice versa. Edges are labeled with known work on the matter. Relations without labels have not been explicitly highlighted in the literature.

**Boot time** Small boot times are an important capability of operating systems to meet the expectations of elasticity and agility of cloud services [67]. Previous work in the literature, for instance, leveraged the small boot times of unikernels to provide just-in-time instanciation of networked services [65]. In addition to the intrinsic boot time of unikernel components, we recognize the following influencing factors:

- Image size: Manco et al. [66] show that VM instanciation times (and therefore boot times) increase linearly with image size due to the overhead of reading the image from storage, parsing it, and laying it out in memory. Other work [71] emphasize that cloud systems need to transfer VM images within data centers from repositories to compute nodes. The I/O overhead is therefore proportional to image size.
- Security and hardening features: Security features impact boot time, either by requiring an increased amount of initialization, or simply by degrading

runtime performance at boot time. The first can be illustrated by smimalloc [62], whose free-list randomization feature entails higher initialization costs compared to non-randomized variants. The latter can be illustrated by the Oscar allocator, which performs twice as more system calls at boot time since there are no *fresh* shadows that can be reused yet [24].

- Runtime memory usage: Madhavapeddy et al. [65] show that the amount of guest memory associated with the virtual machine can significantly influence the instanciation time. The authors observe, for instance, that the VM creation time on the Xen hypervisor increases from 650 ms for a 16 MB domain to over a second for 256 MB. Assuming appropriate guest memory sizing, reduced memory footprint allows to reduce domain creation expenses, ultimately decreasing the total boot time.

- Runtime performance: Runtime performance can be traded off for boot time by lazily initializing objects and structures. Mimalloc, for instance, offloads the initialization of internal structures (e.g., segments, thread local heaps) to the first allocation that accesses these structures [62]. The boot time is consequently faster, at the cost of a decreased runtime performance for the first allocations. The other way around, boot time can be traded off for performance by fully initializing the system, potentially filling up caches in advance. tinyalloc, for instance, pre-initializes all block metadata at startup [82].

- Cloud instance type: Mao and Humphrey [67] show that a larger quantity of cloud resources (e.g., primary and secondary storage, CPUs) takes longer to allocate. As a result, cloud instance size and boot time can be negatively correlated. Increased runtime memory usage and/or decreased runtime performance therefore indirectly influence boot time, as those might require service administrators to increase the size of the instance.

Hence, a memory allocator designed for fast boot times has (1) a minimal initialization time, (2) services boot-time allocations with minimal delay, (3) provides overall low runtime memory usage, and (4) presents a negligible impact on the image size. Lightweight allocators are therefore more prone to provide fast boot times. TLSF fulfills these characteristics as a lightweight allocator with constant-time initialization and bounded fragmentation. tinyalloc, on the other hand, pre-initializes all fresh blocks at startup, potentially resulting in a significant overhead depending on the configuration. The boot time behavior of BBUDDY is also difficult to predict, as it does not run in constant time (see Subsection 2.2.4). Finally, we expect that Mimalloc, despite of its lazy initialization strategy, presents a non-negligible initialization overhead because of its large size and complex data structures.

**Runtime memory usage** Runtime memory usage has an increased relevance in situations of high VM density, as is the case with high density TLS termination [66]. Apart from the intrinsic memory consumption properties of the unikernel, we find the following influencing factors:

- Runtime performance: There are well-known trade-offs between memory usage and performance. A generic concept is caching, which trades off memory for performance (e.g., slab caches [39]). Another example is region-based allocation, which offers substantial speedups (up to 44%) at the cost of an increased memory footprint (up to 63%) [15]. The other way around, performance can be traded off for space by maximizing the block placement effort in the allocator to reduce fragmentation (e.g., by applying a best-fit strategy [91]).
- Security and hardening features: Previous research showed that security features can have an important impact on runtime memory usage. The Oscar and DangSan allocators, for instance, present a memory overhead of 61.5% and 140% with the CPU2006 benchmarks, respectively [24]. Another example is Leijen et al. [62], who show that the addition of security features to mimalloc (notably guard pages) increases the peak memory usage by 71% in their Redis benchmark.

Overall, a good memory allocator with regard to memory consumption features a low fragmentation, a small resident size, and a low degree of bookkeeping, possibly involving a decreased degree of runtime performance. Hardening features should be chosen pragmatically as part of the risk management process, as they also impact the memory footprint. Note that not only low fragmentation but also *bounded* fragmentation can be required in the context of real-time systems [69]. We expect that TLSF presents a low memory footprint due to its good fit mechanism [69]. On the other hand, tinyalloc might present more fragmentation issues due to its first fit mechanism, despite of a low degree of bookkeeping. We expect that BBUDDY presents a high degree of fragmentation as a page allocator (all allocations are rounded up to the next page boundary). Finally, we expect that Mimalloc presents a low degree of fragmentation due to the sharded free-lists, but it tends toward high overall memory consumption on systems that do not support demand paging as the size classes are physically segregated.

**Runtime performance** The following factors typically influence the runtime performance of unikernels:

- Image size: While image size does not influence runtime performance per se, compilation optimizations such as inlining have the capacity to increase performance at the cost of an increased image size [14, 40].

- Runtime memory usage: We have already discussed the trade-offs between runtime performance and runtime memory usage in the previous paragraph: performance can be traded off for memory usage and the other way around.

- Security and hardening features: We have previously illustrated the relation between security and runtime performance in our system call measurements (see 3.1.3), showing that security features such as KPTI multiply the cost of mode switches by x3. A further example is smimalloc, which can present runtime overheads of up to 46% compared to mimalloc [62], the same allocator without security features.

Therefore, memory allocators that optimize runtime performance have inherently good performance properties (fast allocation and deallocation, good locality of reference) and offer an increased potential for inlining. Similarly to runtime memory usage, hardening features should be chosen conservatively as they have a significant impact on runtime performance. Note that not only fast allocation and deallocation but also *bounded* allocation and deallocation times can be required in the context of real-time systems [69]. We expect that Mimalloc, as a state-of-the-art allocator with highly optimized fast and slow paths, provides particularly good runtime performance behavior. Similarly, we expect that tinyalloc provides good performance for workloads that are not deallocation intensive (as deallocations involve an increase of the size of the free-list). Unlike Mimalloc and tinyalloc, TLSF offers bounded allocation and deallocation time, but, set apart pathological cases, runs slower than optimized general-purpose allocators such as the Lea allocator [69].

**Security** Aside from the security features supported by the OS (e.g., ASLR), previous work [19] pointed at the influence of software attack surface (i.e., image size) on the security KPI. Hence, a good memory allocator with regard to security provides security features (e.g., use-after-free detection, free-list randomization, or buffer overflow detection) and has a lean code base to limit the potential for vulnerabilities within the allocator itself. Mimalloc is particularly interesting here since it provides hardening features that can optionally be enabled at build time. TLSF, tinyalloc, and BBUDDY, apart from their small code base, do not provide security features[4]. We note, however, that TLSF has a mature, well-maintained code base, which limits the risks of flaws within the allocator.

**Image size** The image size is an important factor of boot time and security and its relevance increases with the VM density. Unsurprisingly, the image size

[4]In fact, the design of TLSF explicitly assumes a trusted environment [69].

essentially depends on the intrinsic size of unikernel components, the degree of optimizations performed (e.g., dead-code elimination), and the image format (e.g., Multiboot v.s. ELF). Hence, we expect that the memory allocation subsystem can be specialized toward binary size by relying on lightweight allocators. In this regard, we except that BBUDDY (322 source lines of code – SLOC), tinyalloc (418 SLOC), and TLSF (736 SLOC) will exhibit similar results, which will differ from the results for Mimalloc (5868 SLOC).

## 3.3 Conclusion

In this chapter, we have discussed the differences between unikernels and general-purpose operating systems with regard to memory allocation. We showed that kernel memory allocation in unikernels is subject to similar constraints compared to general-purpose OSs (resident memory, physically contiguous memory, etc.). However, we noted that, depending on the unikernel implementation, some of these requirements might be implicitly satisfied. As a result, sharing a single memory allocator in unikernels such as in the current Unikraft 0.4.0 is possible. From the perspective of the application, we highlight the impact of the absence of system calls and the uniprocessor mode.

We supported our claims by measuring the cost of system calls in the Linux kernel, showing their cost has increased by x4.5 with recent Spectre and Meltdown patches — notably kernel page table isolation, widening the gap between system (665 ms) and function (6 ms) calls. Showing that smimalloc performs 25x more `mprotect()` system calls in the espresso workload, we conclude that security-focused memory allocators can benefit from the system call-less architecture of unikernels.

Finally, we discussed how to best leverage memory allocators as a specialization later in unikernels. We presented the typical unikernel KPIs in greater details and how they interdepend. We put our findings in practice, presenting and discussing a list of four memory allocators that are suitable to perform allocator specialization in Unikraft.

# Chapter 4

# Design and Implementation

The previous chapter proposed allocator specialization in the Unikraft framework. We port three memory allocators to Unikraft 0.4.0: Mimalloc [62], TLSF [69], and tinyalloc [82]. We adapt and improve the pre-existing allocator BBUDDY[1]. Finally we proceed to benchmark a wide range of scenarios with off-the-shelf applications, evaluating the potential of allocator specialization in unikernels. This chapter discusses the process of porting the allocators to Unikraft, beginning with an overview of Unikraft's memory allocation subsystem in 4.1. The porting process is described in 4.2.

## 4.1 Memory Allocation Subsystem in Unikraft

Unikraft's allocation subsystem is composed of three layers: (1) a POSIX compliant external API, (2) an internal allocation interface called *ukalloc*, and (3) one or more backend allocator implementations such as BBUDDY.

The external interface is motivated by backward compatibility to facilitate the porting of existing applications to Unikraft. In the case of the C language, the external API is exposed by a modified C standard library which can be nolibc, newlib or musl. The external allocation interface acts as a compatibility wrapper for the Unikraft-specific internal allocation interface, which in turn redirects allocation requests to the appropriate allocator backend. The internal allocation interface therefore serves as a multiplexing facility that enables the presence of multiple memory allocation backends within the same unikernel. Figure 4.1 depicts a simplified `malloc()` call graph in an Nginx Unikraft unikernel, summarizing this layered structure.

[1]See 2.2.4 for an introduction to Mimalloc, TLSF, tinyalloc and BBUDDY.

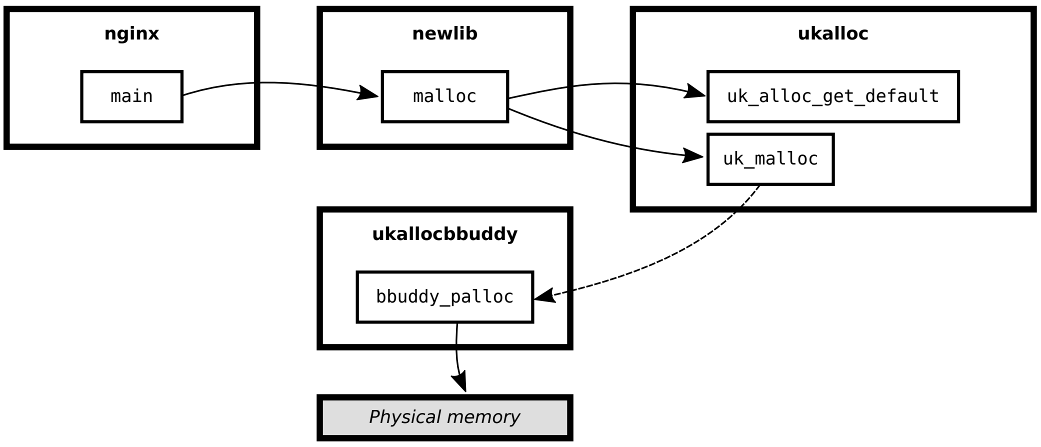


Figure 4.1: Simplified `malloc()` call graph in an Nginx unikernel. Allocation requests transition through three interface layers: the C standard library, the internal allocation interface, and the allocation backend. A complete version of this call graph is available in 4.1.2.

### 4.1.1 Internal Allocation Interface

The internal allocation interface exposes `uk_` prefixed versions of the standard POSIX allocation interface: `uk_malloc()`, `uk_calloc()`, etc. In contrast to POSIX, these functions require the caller to specify which allocation backend should be used to satisfy the request. For instance, `uk_malloc()` is defined as follows:

```
static inline void *uk_malloc(struct uk_alloc *a, size_t size)
```

`uk_malloc()`, like most of the internal allocation interface, is designed as inline method in order to avoid any additional function call overhead in the allocation path[2]. The `struct uk_alloc *` argument represents the allocation backend. This structure essentially contains function pointers that refer to the allocator's implementation of the POSIX allocation interface: `malloc()`, `calloc()`, `posix_memalign()`, etc.

In addition, ukalloc exposes `uk_alloc_get_default()`, which returns the default allocation backend. This function can be used by components that are not assigned a specific memory allocator, such as the C standard library. The default allocator is the first registered allocator.

[2]Enforcing inlining in the Unikraft allocation path is one of the contributions of this thesis.

### 4.1.2 Backend Implementations

Allocators can be implemented as internal or external libraries. They must specify an initialization function which is called at boot time. BBUDDY, for instance, declares the following initialization function:

```
struct uk_alloc *uk_allocbbuddy_init(void *base, size_t len);
```

Initialization functions take a `void *base` pointer to the first usable byte of the heap, along with a `size_t len` argument which specifies the size of heap. Initialization functions must register the allocator to the ukalloc interface via `uk_alloc_register()`. Registering the allocator to ukalloc essentially allows to determine the default allocator, which is the first registered allocator. Initialization functions must fully initialize the allocator. BBUDDY for instance builds up its internal page bitmaps and free-lists. Finally, initialization functions must return a `struct uk_alloc *` that contains function pointers to the allocator's implementation of the POSIX API. The allocator is considered ready to satisfy memory allocations as soon as the initialization function returned.

Initialization functions are called early in the boot process, as many core system components rely on a functioning allocator for their internal structures. We have previously mentioned that several memory allocators can exist concurrently in Unikraft. Which memory allocator is associated to which memory source is handled by the boot process. However, as of Unikraft 0.4.0, there is no mechanism that does this dynamically. The association between memory source and allocator is hardcoded in the boot code.

#### Allocation Interfaces

Dynamic memory allocators in Unikraft are expected to implement the standard POSIX allocation API. Memory allocators used by the kernel must also implement Unikraft's page allocation interface. The page allocation interface is defined as follows:

```
void *palloc(struct uk_alloc *a, unsigned long num_pages);
void  pfree(struct uk_alloc *a, void *addr,
            unsigned long num_pages);
```

`palloc()` allocates a number of pages from the supplied allocation backend. Pointers returned by `palloc()` are always aligned at page size. `pfree()` deallocates pages starting at the given address. Note that the allocator must be the same which has been used for allocation, and the number of pages passed to `pfree()` must match the number of pages initially passed to `palloc()`. Finally, memory allocated using `palloc()` must always be freed using `pfree()` — not `free()`. The page allocation is a low level interface; metadata are outsourced to the caller in order to reduce complexity in the page allocator implementation.

Not all allocators implement the full POSIX allocation interface. BBUDDY for instance only implements the page allocation API. For this reason, ukalloc exposes wrappers that can be used to implement a full POSIX allocation interface on top of a limited set of allocation primitives. In the case of BBUDDY, the page compatibility interface *ifpages* implements the standard POSIX allocation interface on top of `palloc()` and `pfree()`. Figure 4.2 depicts a complete `malloc()` call graph in an Nginx Unikraft unikernel, showing the implementation of `uk_malloc()` on top of `bbuddy_palloc()` by ifpages.

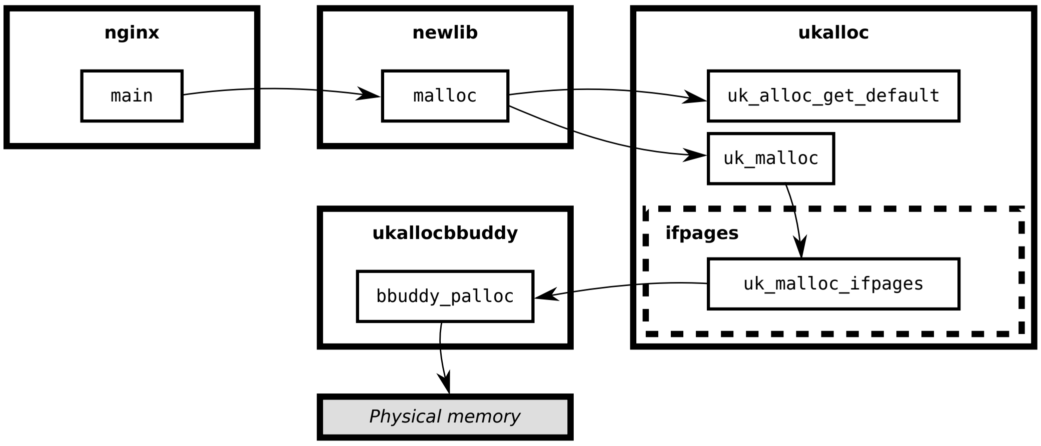


Figure 4.2: `malloc()` call graph in an Nginx unikernel. `uk_malloc()` is implemented by an ifpages wrapper on top of `bbuddy_palloc()`.

## 4.2 Porting Process

We adapt BBUDDY and port three more memory allocators to Unikraft 0.4.0: Mimalloc, TLSF, and tinyalloc. This section presents the porting process of these allocators. We first present a general overview of the engineering effort in 4.2.1. Then, we present three challenges that we encountered during the porting process, and how we addressed them in 4.2.2.

### 4.2.1 General Porting Effort

This subsection presents a quick summary of the general porting process of a memory allocator.

1. First of all, the standard library skeleton is set up. Figure 4.3 depicts the Unikraft library directory in the case of TLSF.

2. Following this, the Unikraft makefile is filled with build information. Build information comprises the allocator's upstream repository, compilation information such as an enumeration of C implementation and header files, as well as build and link flags. Using the link to the allocator's upstream repository, the build system automatically downloads the compressed allocator source and patches it as part of the build process.

3. The Unikraft config file is then filled with dependency information. In the case of Mimalloc, for example, a dependency on the pthread library is declared. A description of the allocator is provided for Unikraft's menuconfig entry.

4. Then, the header file of the library is implemented, which exports the library's public interface. It is straightforward, as only the initialization function is exposed to the rest of the unikernel. The actual POSIX allocation API is accessed via function pointers provided by the initialization function, as described in 4.1.2.

5. Finally, the glue code is implemented, together with required patches for the allocator source. The glue code contains the initialization function, and compatibility wrappers for the ported allocator source.

Compatibility wrappers and patches can be of varying complexity. A basic example is that allocators tend to rely on static variables to store allocator state metadata. This is an issue in the context of Unikraft, as this makes the support for multiple instances of the same allocator within the same unikernel impossible.

```
./
├── Config.uk
├── glue.c
├── include/
│   └── uk/
│       └── tlsf.h
├── Makefile.uk
└── patches/
```

Figure 4.3: Unikraft library directory in the case of TLSF. The port skeleton comprises Unikraft build data, glue code, and patches for the allocator source.

Changing this can vary from straightforward (tinyalloc) to fairly complex (Mimalloc). More fundamental issues can also arise, such as incomplete interfaces, or dependencies that can't be satisfied at boot time. The following subsection presents three of these intricacies, and how we addressed them.

### 4.2.2 Challenges of the Porting Process

Previous subsection presented the general porting process of memory allocators in Unikraft. Unfortunately, memory allocators are both complex systems and core OS components, resulting in various complications of the porting process. This subsection discusses three of them.

#### API Consistency

We have previously mentioned compatibility interfaces: some allocators do not implement the full POSIX interface. In the case of this thesis, neither TLSF nor tinyalloc implement aligned allocation, and none of TLSF, tinyalloc and Mimalloc implement the page allocation interface. It was therefore part of the porting process to develop two compatibility interfaces:

- *ifmalloc* (malloc compatibility interface): implements a POSIX interface on top of a simple `malloc()` and `free()` interface.
- *compat*: implements Unikraft's page allocation interface on top of a POSIX interface.

The implementation of the compat interface is straightforward: `palloc()` can be very easily implemented on top of `posix_memalign()`, without additional metadata. The ifmalloc interface, however, is more challenging. Figure 4.4 summarizes the design of external interfaces, compatibility wrappers, and allocator interfaces in Unikraft's memory management subsytem.

Our first implementation of ifmalloc based on the design of the pre-existing ifpages interface. However, we found that ifpages's `posix_memalign()` wrapper only supported alignment up to a page and returned out-of-bounds pointers in several cases. We therefore needed to develop a new solution for both ifpages and ifmalloc. The following paragraphs summarize our work on this matter.

**Strategy** We overallocate memory from the underlying allocator, and return a block of memory located within the overallocated buffer that meets alignment constraints. Two aspects have to be considered in this design. First, the `free()` function has to be able to recover the original pointer before passing it on to the underlying allocator. Second, the amount of overallocated memory should be chosen carefully in order to waste the least memory possible.

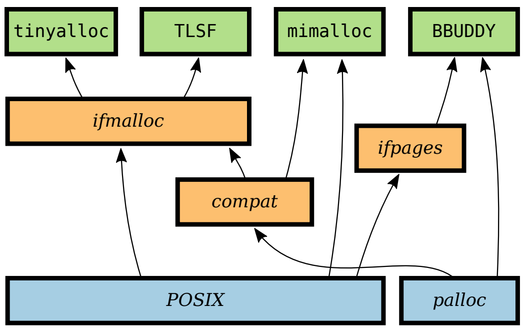


Figure 4.4: Schematic representation of allocator interfaces, compatibility wrappers and external interfaces. tinyalloc and TLSF rely on *ifmalloc* to provide a complete POSIX interface. tinyalloc, TLSF and Mimalloc rely on *compat* to provide a page allocation interface. BBUDDY relies on *ifpages* to provide a complete POSIX interface. ifmalloc and compat have been developed as part of this work. ifpages was redesigned.

**Policy** We apply the same strategy for both ifpages and ifmalloc. However, our implementation differs slightly between them. In particular, the ifpages interface leverages the fact that `palloc()` guarantees page-aligned memory. The following paragraphs present the functioning of the newly designed ifpages interface.

We solve the deallocation problem by introducing new metadata: a pointer to the base address returned by the underlying `palloc()`, and the number of pages allocated (required by `pfree()`). Finding the minimal number of pages that can be allocated without over- or underflowing is not trivial. We minimize memory waste by differentiating between three specific cases, depending on the requested alignment $a$, the page size $p$, and the size of metadata, which is here simplified to 16. Both $p$ and $a$ are powers of two. In the case of `malloc()`, we consider $a := 1$. The buffer size requested by the user is referred to as $s$, and the address of the buffer handed out by the underlying `palloc()` is referred to as $x$. Figure 4.5 represents the memory layout for each of the three cases.

1. $a > p$

The requested alignment is a multiple of the page size. We allocate $a+s$ bytes. Allocating $a$ bytes allows us to be certain to find at least one pointer aligned at $a$ within the chunk handed out to us by `palloc()`. Hence, allocating $a + s$ bytes allows us to be certain to find at least one buffer of size $s$ bytes aligned at $a$ within the chunk.

We store metadata at the beginning of the page preceding the pointer returned to the user. We always return the first pointer aligned at $a$ within the chunk handed out to us by `palloc()`, with one exception: if $x$ was already aligned at $a$, then $x + a$, the second pointer aligned at $a$, is returned. Indeed, returning $x$ would lead us to underflow when storing metadata. Note that $x+a$ is still located within the first $a$ bytes of the chunk, there is therefore no risk to overflow. In conclusion, $a + s$ is the minimal amount of memory that can be requested from `palloc()` in this case.

2. $16 \leq a < p$

   The requested alignment is smaller than a page but larger than 16. We also allocate $a + s$. Since $x$ is aligned at $p$, it is also aligned at $a$, and $x + a$ is the next pointer aligned at $a$ after $x$. $x + a$ is returned and metadata are stored at $x$. Note that $16 \leq a$, so storing metadata will not underflow. $a + s$ is therefore the minimal amount of memory that can be requested from `palloc()` in this case.

3. $a < 16$

   The requested alignment is smaller than the size of metadata. Because of this, allocating $a + s$ would leave insufficient space to store metadata. Note that since 16 is a power of two, it is also a multiple of $a$, i.e. any pointer aligned at 16 will also be aligned at $a$. Hence, we allocate $16 + s$, store metadata at $x$ and return $x + 16$.

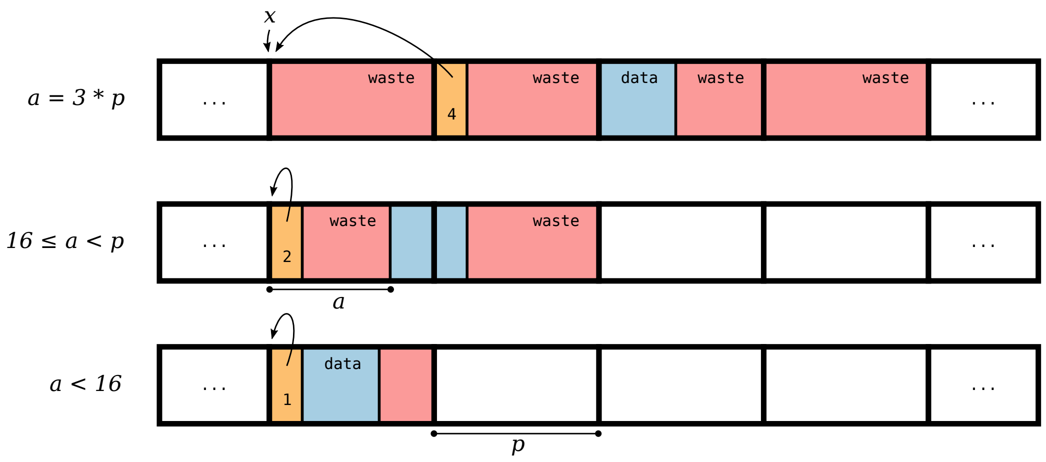


Figure 4.5: Memory layout for various cases of aligned allocation. The first case represents a situation where $a = 3 \cdot p \geq p$. We allocate $3 \cdot p + s$, rounded up to four pages. The second case represents a situation where $16 < a < p$ and $a + s > p$. We therefore allocate two pages. The third case represents a situation where $a < 16$ and $16 + s < p$. We allocate a single page.

**Evaluation** We have implemented a fully POSIX-compliant interface on top of the page allocation interface. Even if this design minimizes memory usage, such wrappers are not memory efficient per se. For allocations above a page, the internal memory usage is $max(\{\frac{s}{16+s}, \frac{s}{a+s}\})$, which falls under 50% as soon as $s < a$. The memory waste problem could be addressed by relying on allocators that propose a full implementation of the POSIX interface such as Mimalloc, or by modifying allocators themselves to implement `posix_memalign()` natively. Both solutions were not practicable in our case since we wanted to evaluate tinyalloc, TLSF and BBUDDY without interfering with their internal functioning. Finally, this wrapper is problematic from a security standpoint as allocator metadata is not separated from user data. Any overflow or underflow could lead to takeover of the allocator. This is not fundamentally an issue in the case of tinyalloc, TLSF and BBUDDY as these allocators do not implement any security counter-measure.

### Allocator Initialization Time

We have previously mentioned that memory allocators in Unikraft are initialized early in the boot process. This is necessary since many core OS components such as the interrupt handler rely on dynamic memory allocation. This, however, posed an issue when porting Mimalloc, which requires a functioning pthreads environment. Initializing pthreads before the allocator is not possible, since pthreads itself requires a functioning memory allocator. Note that this challenge is not specific to Unikraft. Other OSs such as Linux rely on a boot allocator to bootstrap the system in a state that allows more complex allocators to take over [39]. Our solution is not fundamentally different from them: we introduce a minimalist bootstrap allocator which handles allocation requests until the system is ready to initialize Mimalloc, or any other complex allocator. The following paragraphs introduce our solution, how it differs from other approaches, its benefits, and drawbacks.

The bootstrap allocator is initialized by the boot process, as described in 4.1.2. Mimalloc's initialization function in fact initializes the bootstrap allocator. The bootstrap allocator is very similar to a *region*: allocations are performed by linearly increasing a pointer without storing metadata. This promises high performance for early boot time allocations by eschewing bookkeeping[3]. However, unlike region-based allocators, the bootstrap allocator does not support freeing at all. This is not a problem because we observed that allocations done during the early boot time are typically never freed until OS shutdown: interrupt handlers, schedulers, stacks, thread control blocks and TLS areas (for the main and idle threads), PCI device driver structures and long lived pthread internal structures

[3]See 2.2.3 for a more detailed introduction to region-based memory allocation.

are all necessary to the functioning of the unikernel. This region-based design is a noticeable difference to Linux's boot memory allocator, which relies on a first-fit algorithm [39]. Unlike Unikraft, the Linux kernel can free a considerable amount of memory allocated from the boot memory allocator, thus motivating the use of a full-fledged boot memory allocator.

The transition from the bootstrap allocator to Mimalloc is handled by the bootstrap allocator itself. At each allocation, the bootstrap allocator verifies the state of the system. If Mimalloc can be initialized, the transition is started. Note that this can be done efficiently by inspecting the thread control block (TCB), which can be retrieved by bitmasking the stack pointer. During the initialization of Mimalloc, the bootstrap allocator continues to satisfy allocation requests, as Mimalloc's initialization itself might trigger allocation requests via pthreads. The transition function replaces the function pointers in Mimalloc's `struct uk_alloc` to point to Mimalloc's actual interface. This transition is therefore invisible for the rest of the system. Figure 4.6 depicts a `malloc()` call graph at boot time, showing the system state verification, and the incrementation of the internal region pointer.

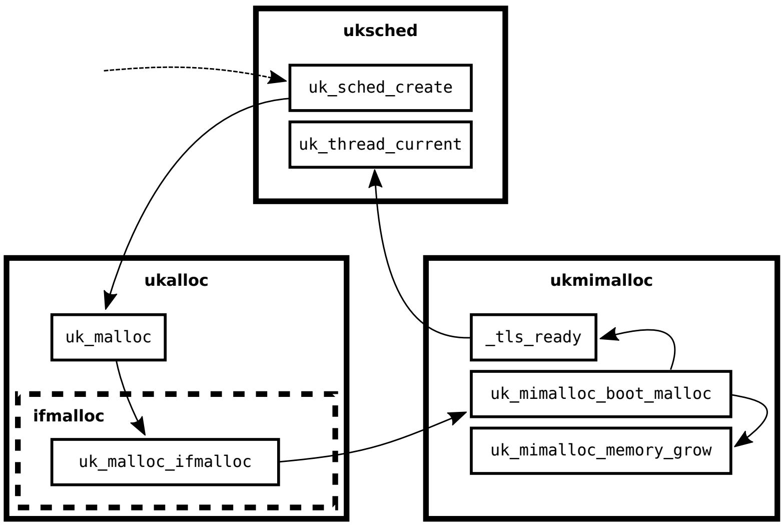


Figure 4.6: `malloc()` call graph at boot time. The state of the system is verified by `_tls_ready()`, retrieving the TCB using `uk_thread_current()`. In the case of this call graph, the system is not yet set up. The internal region pointer is incremented via `uk_mimalloc_memory_grow()`. The bootstrap allocator only implements `malloc()` (and a dummy `free()` function), and therefore uses the ifmalloc interface for POSIX compliance.

**Evaluation** We have implemented a bootstrap allocator that satisfies early boot time allocation requests with minimal overhead. This allows the system to set up before more complex allocators such as Mimalloc can be initialized.

This solution is motivated by the same question as Linux's boot memory allocator [39]. However, unlike Linux, the bootstrap memory allocator does not support deallocation. This is not necessary in the case of Unikraft, because allocations done during the early boot time are only freed at shutdown. If Unikraft would come to evolve and start freeing some of these buffers during runtime, the inability to free memory could become a drawback and an allocator similar to the Linux boot memory allocator could be considered. However, even in this case our minimalist allocator could be considered, offering the opportunity to specialize toward boot time at the expense of memory usage.

### System Call Dependency

Memory allocators manage a pool of memory which can be either statically or dynamically sized. Allocators relying on a fixed size memory pool such as tinyalloc and TLSF are common in embedded systems, and more generally in the context of physical memory management. On the other hand, user space allocators such as Mimalloc typically rely on dynamically sized memory pools. Memory allocators that manage a statically sized memory pool can easily be ported to Unikraft. 4.1.2 describes their initialization in greater detail. In contrast, memory allocators that manage a dynamically sized memory pool present more difficulties, as they typically expect functioning `brk()` or `mmap()` system calls[4]. This requirement is problematic, because it supposes the existence of an underlying logic for physical memory management. However, in the case of a one-to-one memory mapping, the allocator manages both virtual and physical memory, and such physical memory managers therefore do not exist.

**Mimalloc** This is precisely the case of Mimalloc. Unlike TLSF or tinyalloc, Mimalloc's initialization function does not take a memory buffer. Instead, Mimalloc lazily requests memory from the operating system as part of the heap and segments initialization process. Mimalloc does it via `mmap()` system calls on POSIX-like OSs. In addition to this, Mimalloc also supports WebAssembly, which presents an API highly similar to the `sbrk()` interface[5].

[4]See 2.2.1 for an introduction to memory management system calls in general-purpose OSs.

[5]WebAssembly presents a linear memory model with two memory management instructions: `memory.size` which returns the current size of memory, and `memory.grow` which grows the linear memory by a given delta and returns the previous memory size [79].

**Solution** Based on this WebAssembly interface, we implement a simple wrapper which exposes an interface similar to `sbrk()` to Mimalloc. We were motivated to expose this interface instead of `mmap()` because the amount of code that needed to be modified within Mimalloc would have been significantly higher by relying on `mmap()`. This is because Mimalloc's `mmap()` logic expects a full-fledged system call API from the operating system (`madvise()`, `mprotect()`, etc.).

Furthermore, this solution can be tightly integrated with the bootstrap allocator design that we introduced earlier in this subsection. The bootstrap allocator satisfies early boot time allocation by incrementing a region pointer. Whenever Mimalloc is initialized, the bootstrap allocator keeps on managing the memory and supplies Mimalloc via a `sbrk()` API. Figure 4.7 depicts a malloc call graph with Mimalloc upon requesting more memory via the `sbrk()` wrapper.

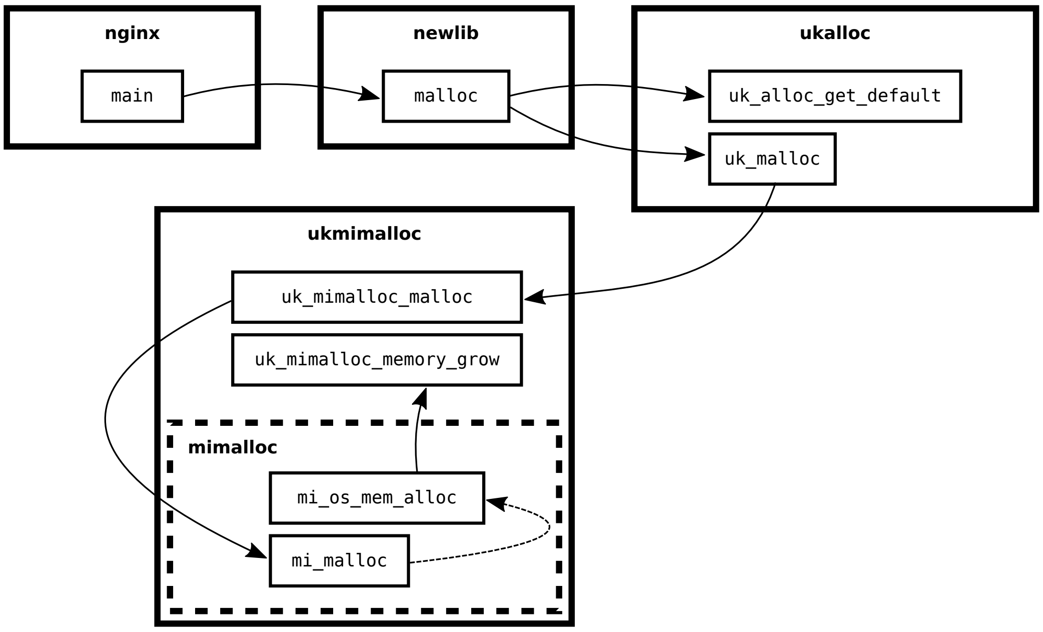


Figure 4.7: `malloc()` call graph with Mimalloc requesting more memory from the `brk`-like wrapper.

**Evaluation** The previously described approach allows to run user-space allocators that rely on the `brk()` system call interface on Unikraft with minimal modification. A similar interface could be developed for the `mmap()` system call. A more contained solution would be to directly patch the allocator to rely on a fixed-size buffer. However, this solution was not practicable for this thesis since we wanted to evaluate possibly unmodified allocators.

## 4.3 Conclusion

In this chapter, we ported and adapted four memory allocators to the Unikraft framework. We showed that while the general porting process of libraries in Unikraft is straightforward, the port of allocators raises non-trivial issues such as maintaining and implementing a consistent allocation interface across allocators, adapting user-space allocators to work in an environment that does not offer the `brk`/`mmap` interface and satisfying their dependencies at boot time.

# Chapter 5

# Evaluation

In the last chapter, we ported several general-purpose memory allocators to Unikraft 0.4.0. In this chapter, we evaluate the performance of these allocators across a wide range of scenarios with off-the-shelf applications: the SQLite database engine, the Redis cache and message broker, and the Nginx web server. We measure the performance gain that can be obtained by specializing the allocator from the perspective of four of the key performance indicators (KPIs) previously introduced in Chapter 3: image size, boot time, runtime performance, and runtime memory usage. This chapter is structured as follows: Section 5.1 introduces our evaluation environment and determines the measurement overhead of Unikraft's tracepoint system. Then, Section 5.2, Section 5.3, and Section 5.4 present our results on image size, boot time, and runtime performance. Section 5.5 concludes with our results on runtime memory usage.

## 5.1 Evaluation Setup

Following measurements are realized on a GNU/LINUX Debian Buster system equipped with 1x Intel® Xeon® E5-2690 v4 with 2.6Ghz base frequency and an invariant timestamp counter (TSC). We use the Linux kernel from the official Debian repositories, version 4.19.0. Hyper-threading is disabled. We use `taskset` to pin the QEMU guest, the QEMU host process, and, if applicable, the benchmark client on physical CPU cores that are isolated from the Linux host scheduler via the `isolcpu` mechanism. Note that the QEMU host process and the QEMU guest are pinned on different CPU cores. While pinning both processes on the same core increases the potential for sharing cache entries, we find that this also increases the contention for CPU resources, leading to significantly reduced performance under certain workloads (e.g., nginx). In the guest, file systems are exclusively in-memory (ramfs).

### 5.1.1 Unikraft Network Stack Configuration

Our first network-bound measurements presented low performance and little variation between allocators. We found that this was due to a performance bottleneck in the network stack. We address these issues by enabling memory pools in Unikraft's LwIP network stack, increasing the maximum number of TCP sockets (which is set to 5 in the default configuration) and the maximum number of TCP listeners. Our optimized LwIP configuration can be found in Appendix B.

### 5.1.2 Runtime Overhead of the Tracepoints System

Several experiments in this chapter rely on Unikraft's tracepoints system, which has been previously introduced in Subsection 2.1.2. We noted that the tracepoints system is associated with a specific runtime overhead due to disabling and re-enabling interrupts, several writes to memory, pointer arithmetic, and a read of the timestamp counter. In order to precisely quantify this overhead, we measure, using the TSC, the time it takes to perform one million tracepoint writes without arguments. The tracepoint label is 10 bytes long. The experiment is repeated 10 times and average values are taken. We perform the same experiment with one 32-bit integer argument, since this specific case is also present in our measurements. Table 5.1 presents our results. We observe that the runtime overhead of a tracepoint write averages 30 ns, which, in the following measurements, will be considered negligible unless otherwise specified.

| Arguments | Tracepoint overhead (in ns) |
|---|---|
| 0 | 30 |
| 1 | 31 |

Table 5.1: Runtime overhead of a tracepoints write with zero and one arguments, in nanoseconds. The addition of one 32-bit argument causes an additional overhead of 1 nanosecond.

### 5.1.3 Baseline Memory Allocator

In order to baseline our binary size and boot time measurements, we implemented *bootalloc*, a minimalist allocator similar to the bootstrap allocator presented in Subsection 4.2.2. The initialization time of our baseline allocator is minimal, as it simply allocates a `struct uk_alloc` at the beginning of the heap, and saves the `base` and `len` arguments. As a region-based allocator, allocation functions simply increment a pointer without bookkeeping information. Deallocation is not supported[1]. Allocation and deallocation functions are therefore as fast as possible [15]. Note, however, that the locality of reference is not necessarily optimal. Achieving optimal locality of reference would require a significantly more complex baseline allocator, e.g., performing heap layout optimizations [81].

## 5.2 Image Size

We have motivated the importance of image size as a key performance indicator in 3.2: small cloud images are necessary to reduce unikernel provisioning time and virtual machine (VM) startup time. In this section, we measure the impact of dynamic memory allocators on the image size of unikernels. We build helloworld, SQLite, and Nginx unikernels for the KVM platform with varying allocators. Images are compiled with optimization for performance (`-O2`). We perform two measurements, enabling and disabling link-time optimizations (LTO) and dead-code elimination (DCE). We compare the size of resulting images using the `du` utility in order to avoid incorrect measurements due to sparse images.

### 5.2.1 Experimental Results

Table 5.2 presents our results. We observe that, concerning the image size, tinyalloc, BBUDDY, and TLSF are indistinguishable from our baseline allocator. For each application, the binary size of all four allocators is identical, with and without LTO and DCE. This surprising result is due to the Multiboot image layout [72] used in KVM: even if all three libraries have different compiled sizes, the differences are hidden by internal alignment.

Mimalloc, on the other hand, presents more contrasted results. Without LTO/DCE, Mimalloc produces a helloworld image which is more than 3x larger than the other images. We provide three explanations: (1) Mimalloc introduces additional dependencies in the helloworld image (notably pthread-embedded), (2) Mimalloc has a larger code base compared to other allocators in this benchmark (see Figure 5.1), and (3) the difference is worsened by internal alignment constraints of the Multiboot specification that we previously mentioned. Note that the difference is much smaller for more realistic images such as SQLite and Nginx: around 4% of the final image (65,536 bytes), due to an alignment jump caused by Mimalloc's slightly larger code base.

[1]The lack of deallocation support is an obvious limitation of bootalloc as a general-purpose allocator. This does not, however, impact our image size measurements. In our boot time measurements, we assume that sufficient memory is available to the system (i.e., bootalloc will not run out of memory).

| | helloworld | SQLite | Nginx |
|---|---|---|---|
| bbuddy | 264,360 | 1,575,136 | 1,509,840 |
| mimalloc | 854,408 | 1,640,744 | 1,575,448 |
| tinyalloc | 264,360 | 1,575,136 | 1,509,840 |
| tlsf | 264,360 | 1,575,136 | 1,509,840 |
| bootalloc | 264,360 | 1,575,136 | 1,509,840 |

(a) Without LTO and DCE.

| | helloworld | SQLite | Nginx |
|---|---|---|---|
| bbuddy | 198,824 | 985,456 | 1,051,232 |
| mimalloc | 330,264 | 985,528 | 1,116,840 |
| tinyalloc | 198,824 | 985,456 | 1,051,232 |
| tlsf | 198,824 | 985,456 | 1,051,232 |
| bootalloc | 198,824 | 985,456 | 1,051,232 |

(b) With LTO and DCE.

Table 5.2: Image size of Unikraft helloworld, SQLite, and Nginx unikernels with varying allocators, with and without LTO/DCE. Values are in bytes.

We observe smaller differences with link-time optimizations and dead-code elimination: the use of Mimalloc causes a 1.7x size increase for helloworld. In the case of Nginx, the difference is around 6% (65,608 bytes) due to alignment in the image. Finally, no significant difference can be observed between Mimalloc and other allocators in the SQLite unikernel: the size jump due to alignment is avoided due to fortunate LTO/DCE optimization.

### 5.2.2 Conclusion

We observe that KVM image sizes present a threshold-like evolution behavior due to alignment constraints. For this reason, concerning the image size, relatively small allocators such as TLSF, tinyalloc, or BBUDDY are indistinguishable from our baseline allocator bootalloc. On the other hand, we find that Mimalloc presents a significantly higher impact on the image size: up to 300% on our helloworld image, and 4-6% on more realistic unikernels such as SQLite and Nginx. We show that this is mostly due to additional dependencies (in the case of helloworld) and threshold effects caused by alignment in the unikernel image.

We stress that Mimalloc is described as a *small* allocator in the literature [62] and expect that larger allocators (e.g., jemalloc [33], tbbmalloc [57]) might present an even larger image size footprint due to previously mentioned effects

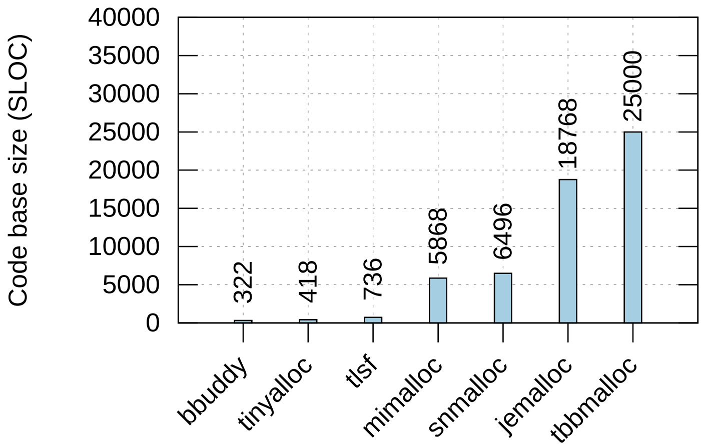


Figure 5.1: Code base size of different allocators, in source lines of code (SLOC).

(see Figure 5.1). Furthermore, our unikernels shipped a single allocator. Unikernels that embed several allocators might be more strongly affected by the image size footprint of allocators. We conclude that while the impact of lightweight allocators (< 1000 SLOC) is negligible, the impact of multiple, or larger allocators can be significant and should therefore be taken into account as part of the specialization strategy.

## 5.3 Boot Time

Subsection 3.2 discussed the impact of dynamic memory allocators on the boot time of unikernels. This section pursues this discussion by experimentally evaluating the impact of allocator choice on this KPI.

In this section, we differentiate between the *internal* and the *external* (or *total*) boot time of unikernels. We define the internal boot time as the time needed to start the application once the virtual machine monitor (VMM) has passed on control to the kernel. In Unikraft, this corresponds to the time needed to transition from the platform entry to the execution of `main()`. In contrast, the external boot time also includes the VMM overhead.

Subsection 5.3.1 presents two optimizations that we implemented in Unikraft in order to enable further boot time measurements. Then, Subsection 5.3.2 evaluates Unikraft's external boot time with different VMMs and motivates allocator specialization toward boot time. Finally, Subsection 5.3.3 evaluates the impact of allocator specialization on internal boot time in the KVM platform.

### 5.3.1 Optimization of Internal Boot Time

First boot time measurements revealed significant issues in the Unikraft framework that made unikernels longer to boot. This effectively masked the impact of memory allocator specialization on boot time. This subsection presents two optimizations that we performed in order to realize our measurements.

#### Calibration of the Timestamp Counter

The timestamp counter (TSC) returns a monotonically increasing value. On processors featuring an invariant TSC, the counter runs at a constant frequency, which is not necessarily the same as the processor's [44]. In order to convert TSC measurements to a value in nanoseconds, the TSC frequency therefore has to be determined. As of Unikraft 0.4.0, the TSC clock frequency is estimated using the i8254 timer over a period of 0.1 s. This solution is undesirable because it delays the boot. Hypervisors advertise the TSC clock frequency via the hypervisor generic `cpuid` timing information leaf [49]. We have patched the Unikraft KVM platform library to retrieve TSC clock frequency via `cpuid` and only fall back to manual calibration if unavailable. Note that the TSC clock frequency is only advertised since QEMU 2.9 and requires the `-cpu +invtsc` command-line argument. These conditions are respected in the following measurements.

#### Optimization of the PCI Bus Driver

First measurements revealed an average boot time overhead of the PCI bus driver of 74 ms, representing more than 98% of the overall boot time. As of Unikraft 0.4.0, the PCI driver implements PCI bus enumeration in a *brute-force* manner, scanning all devices on all buses (256 buses and 32 devices per bus, i.e., 8192 devices in total), ultimately resulting in tens of thousands of I/O read and write operations. Efficient drivers implement PCI bus enumeration in a recursive manner, starting by bus number 0 and further processing downstream bridges [92]. However, refactoring the PCI driver is outside the scope of this thesis. For these measurements, we modified the driver to scan only the first bus. Although this approach does not support PCI bridges, they are not used in our measurements. The overhead of our approach is similar to that of a recursive scan, which would also stop after bus 0. In this way, we reduced the PCI bus driver initialization overhead to less than 400 μs.

### 5.3.2 External Boot Time

We evaluate the external boot time of a Unikraft helloworld unikernel with QEMU/KVM and Solo5 [6, 90], a VMM designed specifically for unikernels. This

| | Total | Hypervisor | BIOS/Firmware | Unikraft |
|---|---|---|---|---|
| QEMU/KVM, NIC | 86 | 37 | 49 | < 1 |
| QEMU/KVM | 76 | 35 | 41 | < 1 |
| Solo5 | 3.4 | n.a. | n.a. | n.a. |

Table 5.3: Average external boot time with (1) QEMU/KVM 5.0.50 and SeaBIOS 1.12.1 with and without NIC and (2) Solo5 0.4.1, in milliseconds (ms).

measurement is important to understand the influence of the internal boot time on the total boot time, that is, the potential of allocator specialization on this KPI. The measurements are realized using the `perf` program. In the case of QEMU/KVM, we leverage custom tracepoints to further separate the overhead of the hypervisor from the overhead of the BIOS [35]. We realize two distinct measurements, one with and one without network interface controller (NIC). Instrumenting the Solo5 platform is a non-trivial task. Therefore, our Solo5 measurements include boot, execution, and destruction time, and thus provide an upper bound for the external boot time. The VM is assigned 2 MiB of guest memory. The experiment is repeated 100 times and average values are taken. Table 5.3 presents our results.

These measurements show that the unikernel itself represents a minor fraction of the boot time overhead in QEMU/KVM. The overhead is primarily divided between the initialization of QEMU/KVM and the BIOS, which is slightly more expensive. Solo5 reduces the numbers significantly with total boot times averaging 3-4 ms. This highlights the large amount of efforts invested in VMMs to reduce boot time. As a point of reference, `fork()` or `exec()` system calls typically present overheads of 1 ms [66], and boot times of 30 ms are sufficient to perform just-in-time instanciation [65].

While these measurements provide a first understanding of the VMM overhead, these do not include a realistic unikernel boot time as the loaded application is a simple helloworld. We anticipate that more realistic applications such as Nginx will present longer boot times that will be significant with regard to the total boot time in specialized VMMs such as Solo5[2], thus motivating further optimization of the internal boot time. In the next subsection, we evaluate the internal boot time of unikernels, breaking down to individual components and determining the potential of memory allocator specialization.

[2]Note that Solo5 is not the only VMM that targets lightweight VMs. Amazon Firecracker [10], for instance, is also supported by Unikraft and we observed similar total boot times in the range of a few milliseconds.

### 5.3.3 Internal Boot Time

Subsection 3.2 pointed at two qualities of dynamic memory allocators that can influence boot time: (1) overhead of the initialization function and (2) runtime performance at boot time. We later discussed in Subsection 4.1.2 that initialization functions depend on the amount of memory allocated to the virtual machine at boot time (e.g., via the `-m` command line option in KVM). Therefore, we study the boot time of unikernels not only *vertically* by varying the allocator, but also *horizontally* by varying the amount of memory being allocated to the VM.

We measure the internal boot time of helloworld, SQLite, and Nginx unikernels with varying allocators. The helloworld unikernel is interesting to study as it represents the *base cost*, or lower bound of Unikraft's boot time. Nginx and SQLite, on the other hand, are realistic unikernels with and without networking. Measurements are realized using Unikraft's tracepoints system. We consider the measurement overhead (2 to 9 tracepoint writes) to be negligible since it accounts for at most 0.005% of measured values according to our results from Subsection 5.1.2.

#### Helloworld

We evaluate the boot time of a minimalist helloworld unikernel in KVM. The PCI bus driver and virtio drivers are disabled. The unikernel includes the following main libraries: nolibc, ukalloc, ukargparse, ukboot, ukdebug, and uktimeconv. This corresponds to the minimal set of libraries required to build a functional Unikraft image. Table 5.5 shows our results for total internal boot time. Figure 5.2 shows a breakdown of the unikernel boot time overhead.

First, we observe that BBUDDY does not have a constant initialization time with increasing VM memory. This is due to its initialization function which splits

| | 2 MiB | 512 MiB | 1024 MiB |
|---|---|---|---|
| bbuddy | 68 | 5703 | 6558 |
| mimalloc | n.a. | 352 | 351 |
| tinyalloc | 69 | 70 | 70 |
| tlsf | 48 | 48 | 48 |
| bootalloc | 39 | 40 | 40 |

Table 5.4: Internal boot time of a minimal Unikraft helloworld unikernel with varying allocators and memory sizes, in microseconds (µs). The default configuration of Mimalloc does not support 2 MiB heaps, explaining the missing value. This characteristic is discussed in further details in Subsection 5.5.1.

the linear address space into smaller blocks of size $2^{n+s}$ (see 2.2.4). In this benchmark, we observe that this behavior does not scale for large memory sizes. Considering a memory size of 1024 MiB, a unikernel with BBUDDY boots 19x slower than with Mimalloc, and 164x slower than with bootalloc. Increasing the VM memory from 2 MiB to 1024 MiB multiplies the boot time by x96.

Second, we observe that Mimalloc boots significantly slower than other allocators: 5x slower than tinyalloc, 7x slower than TLSF, and 9x slower than bootalloc. We provide two explanations: (1) Mimalloc is a considerably more complex allocator and (2) the additional dependencies of Mimalloc result in additional initialization time (see Figure 5.2). Unlike other images, the Mimalloc unikernel also initializes the virtual file system (vfscore) and pthread-embedded libraries. Note that the initialization of Mimalloc happens in two steps: (1) the bootstrap allocator is initialized at allocator instanciation time and (2) the initialization of Mimalloc is triggered as part of the initialization of pthread-embedded, as soon as the thread local storage (TLS) has been set up (see Subsection 4.2.2). Therefore, Mimalloc's actual initialization overhead is accounted in Figure 5.2 as part of *pthreads*.

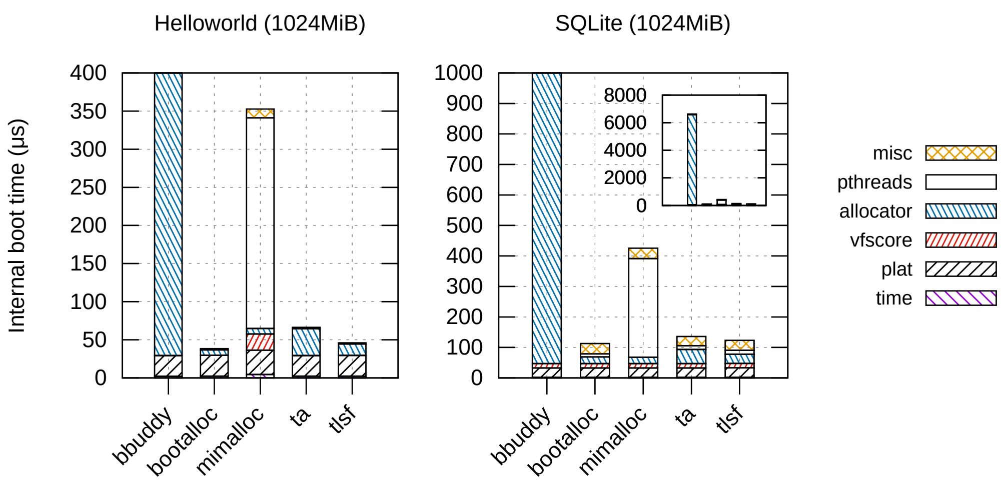


Figure 5.2: Internal boot time of Unikraft helloworld and SQLite unikernels, with varying allocators. The minimal helloworld unikernel and the SQLite unikernel are both passed 1024 MiB of memory. Like previously shown in Table 5.5, Unikraft's page allocator behaves pathologically. The miniplot at the right displays an uncropped version of the SQLite plot. pthread-embedded is initialized in all SQLite images, but this process takes more time in the Mimalloc unikernel since it also includes the transition from the bootstrap allocator to Mimalloc. The initialization overhead of the TSC clock discussed in 5.3.1 is displayed as *time*.

### SQLite

We perform a similar experiment with a Unikraft SQLite unikernel. In addition to the base libraries included in the helloworld image, the SQLite unikernel also initializes vfscore, pthread-embedded, and relies on the newlib C standard library. Figure 5.2 presents our results.

These results confirm that most of the overhead displayed as *pthreads* in the Mimalloc image originates from the initialization of Mimalloc. We further observe that the difference between tinyalloc (136 µs), TLSF (123 µs), and bootalloc (113 µs) is less significant. In the case of BBUDDY and Mimalloc, the allocator still represents the main cost of the unikernel boot process: BBUDDY (7.7 ms) and Mimalloc (425 µs) are 68x and 3.8x slower to initialize than bootalloc, respectively.

### Nginx

Finally, we measure the internal boot time of a Unikraft Nginx unikernel. In addition to the libraries already shipped by the SQLite unikernel, the Nginx unikernel includes virtio network drivers, the Unikraft generic bus interface (*ukbus*), the PCI bus driver, the lwip network stack (without memory pools), and mounts an initramfs disk image (*rootfs*). Figure 5.3 presents our results.

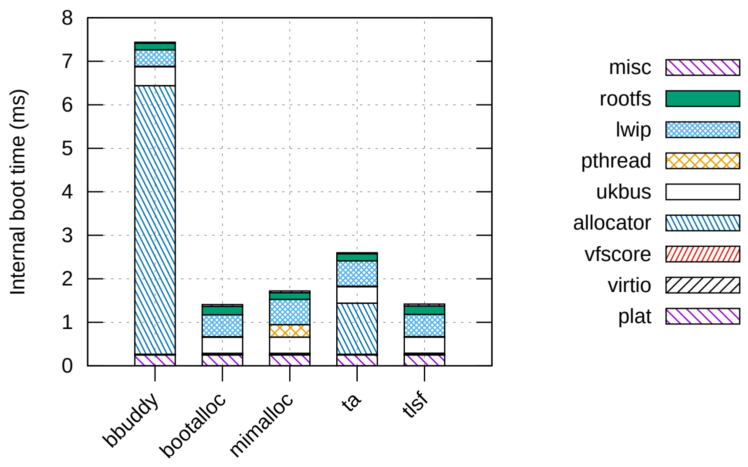


Figure 5.3: Internal boot time of Unikraft Nginx Unikernel, with varying allocator. The initialization of the PCI bus driver discussed in 5.3.1 is included in *ukbus*. The calibration of the TSC is included in *plat*.

| | lwip | $\sigma$ | rootfs | $\sigma$ |
|---|---|---|---|---|
| bbuddy | *380.57* | 2.2 | 153.26 | 1.8 |
| mimalloc | 580.12 | 4.3 | *152.90* | 4.2 |
| tinyalloc | 580.12 | 2.8 | 164.66 | 4.7 |
| tlsf | 510.96 | 3.3 | 192.50 | 4.8 |
| bootalloc | 503.09 | 2.8 | 195.17 | 3.4 |

Table 5.5: Mean initialization time of lwip and rootfs in an Unikraft Nginx unikernel with varying allocators, in microseconds (μs). The standard deviation of our measurements is indicated in the $\sigma$ column.

We find that, even with a real-world kernel that initializes an entire network stack, the scalability issue of BBUDDY is still critical: the initialization of BBUDDY (7.4 ms) remains 4.4x slower than Mimalloc (1.7 ms) and 5.3x slower than TLSF and bootalloc (1.4 ms). These discrepancies are relevant compared to the total boot times observed in the previous subsection: BBUDDY's initialization overhead, for instance, represents 2.2x Solo5's total boot time.

In contrast to previous measurements, tinyalloc has a particularly high initialization time of 2.6 ms (1.5x higher than Mimalloc). We explain this observation by the number of fresh blocks initialized by tinyalloc: 2048 for tinyalloc and SQLite, 200,000 for Nginx. Note that the number of fresh blocks initialized by tinyalloc is a user configurable parameter (see 2.2.4 for an introduction to tinyalloc). It is therefore possible to choose a lower number of blocks. However, we find that a lower number of blocks results in decreased Nginx performance, effectively trading off runtime performance for boot time.

Most components do not exhibit a different runtime profile depending on the allocator. Nevertheless, we observe differences in the initialization time of the LwIP network stack and the decompression and mounting of the initramfs disk, as shown in Table 5.5. Notably, the network stack is consistently 50% faster to initialize with BBUDDY than with Mimalloc and tinyalloc, and 32-34% faster compared to TLSF and bootalloc. On the other hand, BBUDDY and Mimalloc do not exhibit differences during the initialization of the rootfs. Both offer a speedup of 26-27% compared to TLSF and bootalloc, but only 7% compared to tinyalloc. It is unclear why BBUDDY offers such a speedup for both lwip and rootfs. The fact that BBUDDY performs consistently better than bootalloc might indicate a better behavior with regard to locality of reference, however this is hard to verify without measuring the number of cache- and TLB-misses, which Unikraft does not support yet.

### 5.3.4 Conclusion

In this section, we evaluated the impact of allocator choice on boot time. We first studied the external boot time with different VMM setups. Then, we continued with a study of internal boot time with different off-the-shelf applications.

Our study of total boot time highlights that, while classical hypervisors were neither meant for fast boot times nor for unikernels, there exist specialized VMMs such as Solo5 that can leverage the full potential of unikernels to provide minimal boot times. We show that, while classical hypervisors represent the vast majority of the boot time overhead, specialized solutions such as Solo5 effectively reduce the hypervisor overhead to a few milliseconds, thus motivating further optimization of internal boot time.

In our study of internal boot time, we showed that the choice of allocator impacts the internal boot time significantly, even for realistic unikernel applications such as Nginx or SQLite. Coupling boot time optimizations (in particular, TSC frequency calibration) and allocator specialization, we achieve a minimal internal boot time of 39 µs in the helloworld unikernel, 113 µs in the SQLite unikernel, and 1.4 ms in the Nginx unikernel. In addition, we show that the initialization of the allocator can represent a major source of overhead at boot time, notably due to scalability issues. Finally, we observe that allocators can impact the initialization time of core unikernel components such as the network stack.

## 5.4 Runtime Performance

In this section, we measure the impact of allocator specialization on runtime performance in the SQLite database engine, in the Nginx web server, and in the Redis cache and message broker.

### 5.4.1 SQLite

We measure the performance of the SQLite database engine in Unikraft with varying allocators. To this end, we perform an increasing number of SQL `INSERT` queries[3] within an SQLite 3.30.1 Unikraft kernel, a methodology similar to Boicea et al. [16]. Note that no networking or sockets are involved. The database is located in an in-memory file system to minimize I/O overhead. We record the time needed to perform all queries using Unikraft's tracepoints system. The experiment is repeated 30 times and average results are taken. Figure 5.4 presents our results.

[3] We performed similar experiments with different types of queries and found similar results. Nevertheless, our results showed that SQL `INSERT` queries stressed the allocation subsystem best, making them more fit to highlight the differences between allocators.

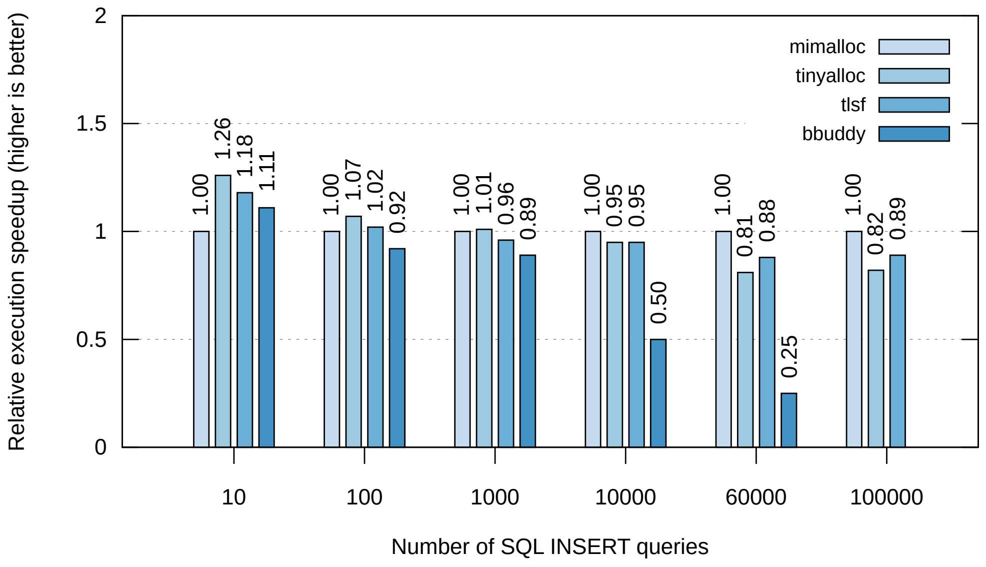


Figure 5.4: Average execution speedup with an increasing number of SQL `INSERT` queries in SQLite Unikraft, relative to Mimalloc. BBUDDY did not run the 100,000 queries benchmark to completion, which explains the missing value. Results are normalized against Mimalloc since (1) it presents values everywhere and (2) we expect that it presents the best results overall as a state-of-the-art allocator.

We observe that tinyalloc offers a speedup of 26% and 7% over Mimalloc for 10 and 100 `INSERT` queries, respectively. On the other hand, Mimalloc performs 18% faster for 100,000 queries. Interestingly, this benchmark also shows a collapse of BBUDDY's performance after 1000 queries, performing 50% slower than Mimalloc for 10,000 queries and 75% slower for 60,000 queries. For 100,000 queries, the BBUDDY unikernel encountered an out-of-memory situation and therefore did not run to completion. The precise cause of this observed performance is discussed later in this subsection. Overall, we observe that lightweight allocators perform best for short workloads, but tend to underperform in the long run.

Note that Mimalloc underperforms for short workloads because of its lazy initialization strategy: whenever a memory block of a certain size class is requested, the corresponding heap, segment, and ultimately page is retrieved (see 2.2.4). If no segment or page was initialized for this size class, Mimalloc appropriately initializes the missing structures. This initialization overhead can be substantial: Table 5.6 shows that Mimalloc undergoes an extra cost of 105 µs during database initialization compared to tinyalloc (x2.6).

On the other hand, Mimalloc's fast allocation and deallocation paths and high locality of reference result in high performance for longer workloads. For 60,000

| Allocator | Opening time (in µs) |
|---|---|
| bbuddy | 104 |
| mimalloc | 171 |
| tlsf | 99 |
| tinyalloc | 66 |

Table 5.6: Average time needed to open an empty SQLite database with varying allocators, in microseconds (µs).

and 100,000 queries, Mimalloc presents an average execution speedup of 11% to 19% compared to TLSF and BBUDDY. This translates to a difference of execution time of more than a second between Mimalloc and tinyalloc (see absolute numbers in Appendix A.1). In order to better explain these results, we perform the same experiment and record the average execution time of `malloc()` for each SQL query, with varying allocators[4]. Figure 5.5 shows our results.

In the case of tinyalloc we observe that the allocation overhead increases with the number of queries. For the first 1000 queries, the time in `malloc()` averages 25 ns, the second fastest after Mimalloc. After 20,000 queries however, the average allocation time exceeds 80 ns, effectively becoming the slowest one. This effect is due to the linear cost of tinyalloc's sequential fit strategy. Appendix A.1 provides average numbers for all queries: overall, tinyalloc has the highest allocation time, confirming that tinyalloc performs best for short workloads but is unsuitable for long-lived applications.

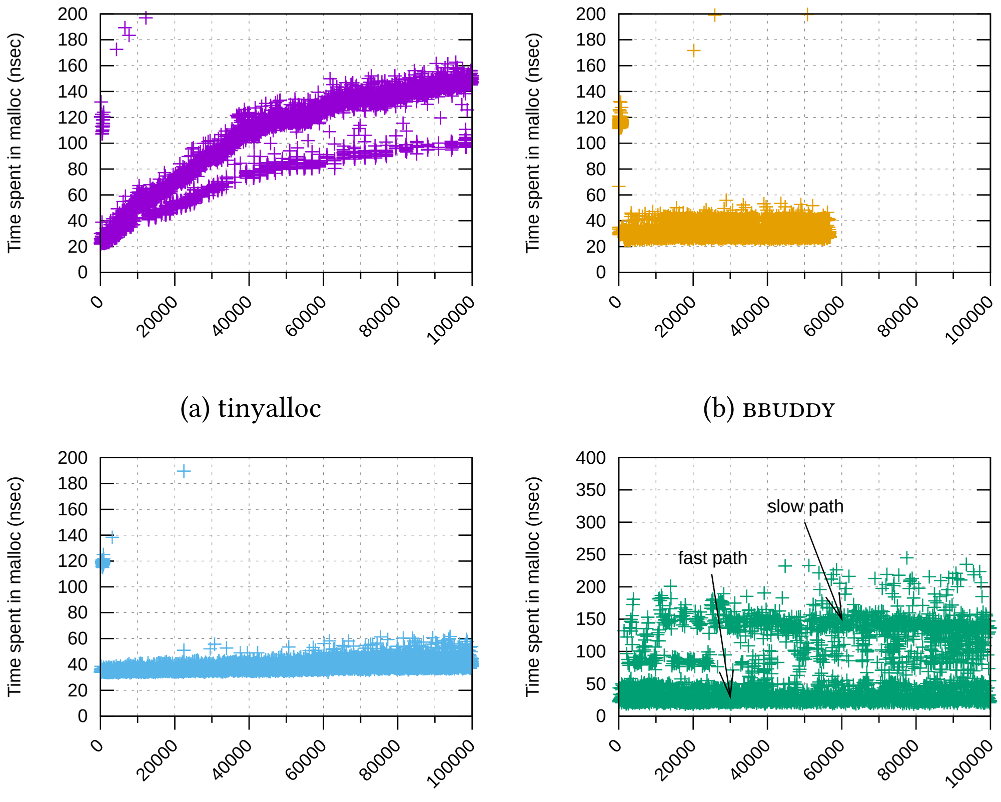
TLSF, on the other hand, exhibits the expected behavior of a real-time memory allocator: the allocation overhead is low (38 ns, the second lowest after Mimalloc) and very stable (considering all queries, the standard deviation is only 4 ns).

The steadiness of TLSF contrasts with Mimalloc, which exhibits two categories of cost: the fast path, which features the overall lowest allocation cost, presenting allocation costs as low as 18 ns, and the slow path, with a considerable number of allocations executing in more than 100 ns. Despite of the significant costs incurred in the slow path, the overall allocations remain fastest of all allocators with an average of 32 ns (18% faster than TLSF).

The presence of two categories of cost also applies to BBUDDY at a different scale. Figure 5.5b (cropped at 200 ns) shows that most allocations average at 30 ns. Figure 5.6, on the other hand, shows that BBUDDY presents tree-like outliers in the order of microseconds. These outliers are caused by the buddy system's splitting strategy. Note that BBUDDY's outliers are not sufficient to explain the overall

[4]We performed the same measurements for `free()` and obtained similar results. These results are not present in this study for space reasons.

underperformance after 1000 queries. We expect that these results are caused by a very low cache locality: since BBUDDY rounds up allocation to the next page boundary, small allocations occupy a full page, entailing a very low locality and subsequent cache and TLB misses. We could not perform this measurement because Unikraft does not support reading the performance registers.



(a) tinyalloc

(b) BBUDDY

(c) TLSF

(d) Mimalloc

Figure 5.5: Average allocation time in SQLite Unikraft with an increasing number of SQL `INSERT` queries and varying allocators, in nanoseconds (ns). A point at coordinate $(x, y)$ indicates that the average execution time of `malloc()` was $y$ ns during the $x$th `INSERT` query. BBUDDY did not run to completion and therefore only presents values until 56,850. Note that the results presented in Figure 5.4 reached 60,000 because the space overhead of the tracepoints system was less important. 5.5b is cropped at 200 ns, effectively hiding a second category of costs. An uncropped graph is visible in Figure 5.6.

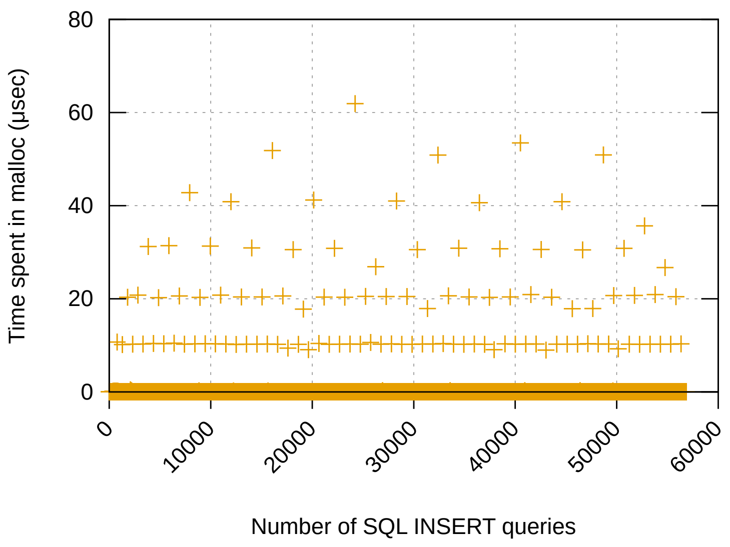


Figure 5.6: Evolution of the average allocation time in SQLite Unikraft with an increasing number of SQL `INSERT` queries and the BBUDDY allocator, in microseconds (µs).

### 5.4.2 Redis

We measure the performance of the Redis cache and message broker in Unikraft with varying allocators using the `redis-benchmark` [5] tool. The benchmarking tool runs on the host and communicates with the Redis unikernel guest via TCP. The Redis database lives exclusively in memory, and snapshots are disabled. Appendix B presents our Redis configuration in further details. Our benchmark opens 30 concurrent connections with a pipelining level of 16[5] and executes a total of 100,000 requests. In order to baseline our results, we perform the same experiment in (1) our bare metal system (described in 5.1), (2) an Alpine Linux virtual machine (KVM) with Linux 5.4.34, and (3) an Alpine Linux Docker container. All experiments are based on Redis 5.0.6. Figure 5.7 presents our results in Unikraft and Figure 5.8 presents our baselining results.

Figure 5.7 shows a clear ordering in allocator performance: For all four request types, Mimalloc performs best, followed by TLSF, BBUDDY, and tinyalloc. We observe that, for `GET` requests, Mimalloc performs 7.5% faster than TLSF, 11.5%

[5]Without pipelining, client and server communicate in a blocking manner: the client makes a query and must wait for the server to answer before making a new query. With a pipelining of 16, the client can send 16 requests at a time (potentially all in the same TCP packet) and receive all answers simultaneously [20]. In the case of our benchmark, the use of pipelining allows us to circumvent potential network bottlenecks, and increase the pressure on the memory allocator in the unikernel, thus providing a better understanding of the differences among allocators.

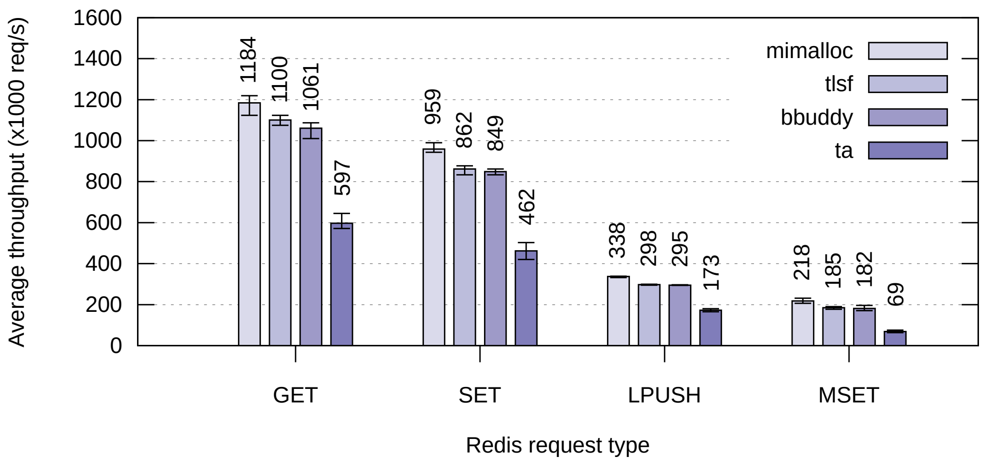


Figure 5.7: Average throughput of Redis Unikraft in thousands of requests per second with varying allocators and request type.

faster than BBUDDY, and 98% faster than tinyalloc. `SET` (set the value of a key), `LPUSH` (insert at the head of a list) and `MSET` (batched `SET` operations) are more expensive operations for the Redis server, inducing an increased pressure on the allocator. The difference between allocators therefore gradually increases: Mimalloc performs 11% faster than TLSF for `SET` requests, 13% faster for `LPUSH` requests, and 18% faster for `MSET` requests. These results are consistent with Leijen et al. [62] who observed a similar behavior with Mimalloc, reporting average speedups of 7% to 14% compared to tcmalloc and jemalloc, respectively.

Our baseline measurements (Figure 5.8) show that the Unikraft unikernel outperforms the native Linux Redis server by 2.4% for `GET` requests, the docker container by 25%, and the Linux VM by 53.5%. Concerning `SET` requests, the Unikraft unikernel outperforms the native Linux server by 13%, the docker container by 28%, and the Linux VM by 56%. The fact that Unikraft outperforms (or performs as well as) a native Redis server is at first startling. We provide the following explanations:

- The Unikraft server performs allocator specialization, while the native Redis server does not. For both `GET` and `SET` queries, the performance of Unikraft with TLSF is below the performance of the native Redis server. Allocator specialization is therefore the enabling element for Unikraft to outperform the native Linux server.
- Note that we did perform the same benchmark replacing the allocator in our native Linux server via the `LD_PRELOAD` mechanism, however, this did

not make the native server perform significantly better. We expect that the difference would be more significant if Mimalloc would be present at compile time instead of relying on the preloading mechanism. Indeed, making the compiler aware of the allocator allows it to perform compile and link time optimizations. Unfortunately, we could not perform this experiment since Mimalloc is not natively supported by the current Redis code base.

- Finally, we recall the high cost of system calls and note that Redis is a particularly system call intensive application. Zhang et al. [95] show that a Redis server can generate as much as 32,700 system calls per second. In order to illustrate the critical impact of system overhead on the performance of Redis, we perform the same baseline experiment on a host system with security mitigations disabled (`mitigations=off` [2], notably disabling KPTI). We find that, in such as setup, a Redis server on native Linux performs 8% *better* than Unikraft for `GET` requests. We also observe that the performance gap between Unikraft and Docker shrinks to 13.6%. For `SET` requests, the performance difference between Unikraft and native Linux is statistically insignificant (approximately 0.005%), and the difference between docker and Unikraft is around 21%.

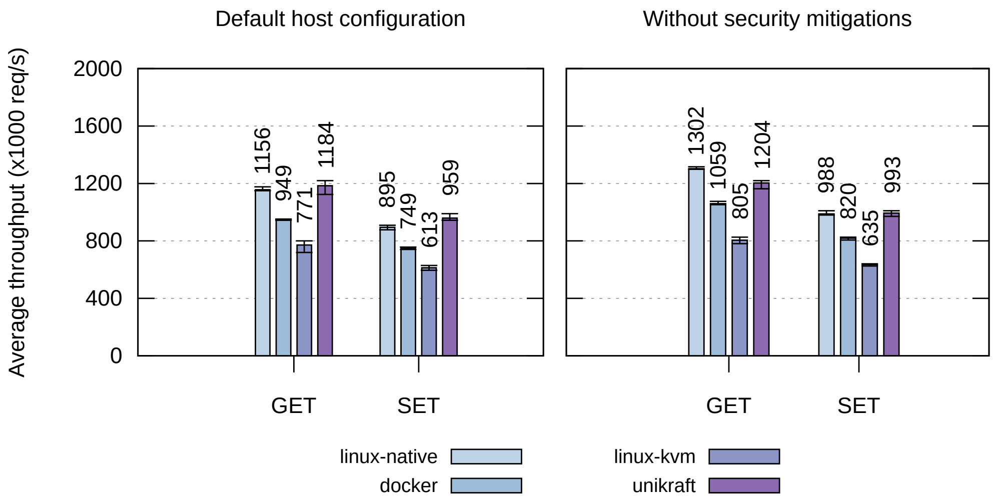


Figure 5.8: Average throughput of Redis Unikraft in thousands of requests per second with varying systems and request type. The diagram at the left presents results with the default Linux configuration. The diagram at the right presents results with security mitigations disabled (`mitigations=off`).

### 5.4.3 Nginx

We measure the performance of the Nginx web server in Unikraft with varying allocators. In order to limit the impact of I/O calls on our results, logging is disabled. Appendix B presents our Nginx configuration in further details. Using the `wrk` [8] HTTP benchmarking tool, we benchmark the server for 1 minute with 14 threads and 30 connections from the host. A single static file of 612 bytes (a small HTML page) is requested. The experiment is repeated 10 times and average values are taken. In order to baseline our results, we perform the same experiment in (1) our bare metal system (described in 5.1), (2) an Alpine Linux virtual machine (KVM) with Linux 5.4.34, and (3) an Alpine Linux Docker container. All experiments are based on Nginx 1.15.6. Figure 5.9 presents our results in Unikraft, and Table 5.7 presents our baselining results.

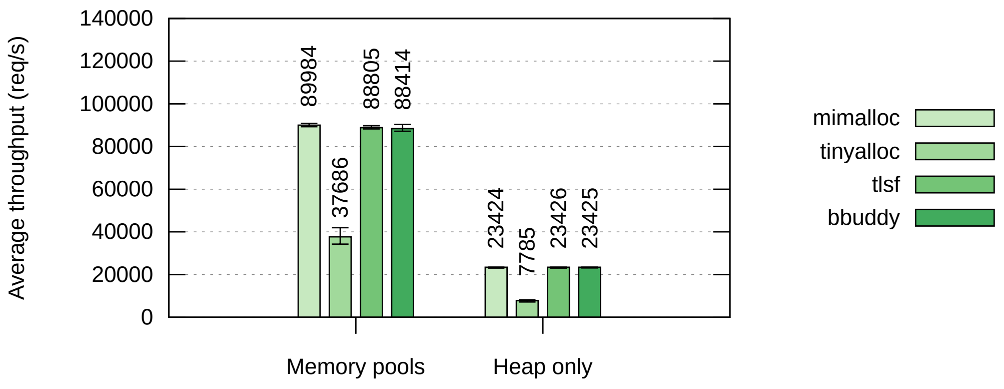


Figure 5.9: Average throughput of Nginx Unikraft in requests per second, with varying allocators, with and without LwIP memory pools.

Figure 5.9 shows that the difference between TLSF, BBUDDY, and Mimalloc is around 1%, which is equivalent to the uncertainty of our results. There is therefore no statistically significant difference between them. On the other hand, tinyalloc underperforms by 39% compared to TLSF, BBUDDY, and Mimalloc: tinyalloc's sequential fit mechanism does not adapt very well to long lived workloads such as web or database servers, where the free-list tends to reach large sizes.

The absence of statistical difference between TLSF, BBUDDY, and Mimalloc is unexpected. Nevertheless, these results are consistent with Larson and Krishnan [60] who show that reasonably well-designed allocators present little difference in web servers on uniprocessor systems. The authors highlight that critical characteristics of memory allocators in the context of server applications are (1) scalability on systems that support symmetric multiprocessing (SMP), (2) thread independence, (3) stability, and (4) predictable performance. Since Unikraft works exclusively in uniprocessor mode, scalability on SMP systems and thread independence are not considered here. We expect that this experiment will exhibit more differences between allocators once SMP support will be added to Unikraft.

The performance speedup granted by custom allocation in the network stack averages x3.8 for TLSF, BBUDDY, and Mimalloc. Note that, in heap mode, the network stack tends to perform allocations and deallocations at a high rate, resulting in a considerably larger free-list and overall degraded performance for tinyalloc. As a result, the memory pool speedup is slightly higher for tinyalloc (x4.8).

Finally, our baseline results (Table 5.7) show that Unikraft performs 23% faster than the Docker container, 53% faster than the general-purpose Linux VM, but 11% slower than the native Nginx web server.

| System | Platform | Throughput (in req/s) |
|---|---|---|
| Debian Linux | Native | 100,223 |
| Unikraft (Mimalloc) | KVM | 89,984 |
| Alpine Linux | Docker | 73,063 |
| Alpine Linux | KVM | 58,858 |

Table 5.7: Average throughput of Nginx in requests per second, with varying systems and platforms.

### 5.4.4 Conclusion

We show that allocator specialization can improve performance significantly. In particular, we observe speedups up to 26% for an SQLite unikernel using tinyalloc and up to 18% for a Redis unikernel using Mimalloc. Nginx mitigates our results, showing no significant difference between allocators (in fact, Nginx presents a clear *loss* of performance with tinyalloc). We support our results with microbenchmarks showing that the time spent in the allocator accounts for a significant part of these differences. In addition, our study of the SQLite unikernel confirms that allocator specialization strategies should not only consider the application but also the targeted workload: while SQLite exhibits speedups of up to 26% for short workloads (< 1000 queries), Mimalloc offers a clear speedup of >10% for long-lived workloads (1000 - 100,000 queries).

## 5.5 Runtime Memory Usage

In this section, we measure the impact of dynamic memory allocators on runtime memory usage. To this end, we determine the peak memory consumption of Unikraft SQLite, Nginx, and Redis images under different workloads for the KVM platform.

### 5.5.1 SQLite

We perform the same experiment as previously realized in Subsection 5.4.1: we perform an increasing number of SQL `INSERT` queries in a Unikraft SQLite unikernel. In order to determine the minimum memory usage, we (1) leverage tracepoints to signal successful execution of all queries and (2) determine the minimum guest memory size (passed via `-m` for KVM) that allows the unikernel to reliably run to completion. Table 5.8 presents our results.

| | 10 | 100 | 1000 | 10,000 | 60,000 | 100,000 |
|---|---|---|---|---|---|---|
| bbuddy | 5 | 6 | 20 | 162 | 946 | n.a. |
| mimalloc | 261 | 261 | 261 | 261 | 261 | 261 |
| tinyalloc | 14 | 14 | 15 | 19 | 42 | 61 |
| tlsf | 4 | 4 | 4 | 9 | 37 | 59 |

Table 5.8: Memory usage of SQLite Unikraft, with varying allocators and number of requests, in mebibytes (MiB). BBUDDY did not run to completion within the 1 GB limit for 100,000 queries.

As a page allocator, BBUDDY suffers from a high degree of internal fragmentation. While its space usage is comparable to that of TLSF or tinyalloc for less than 1000 queries, BBUDDY requires 18x and 25.5x more memory than TLSF for 10,000 and 60,000, respectively. For 100,000 BBUDDY does not run to completion, indicating that its space usage exceeds the limit of 1 GB imposed by Unikraft's one-to-one memory mapping.

Mimalloc allocates 256 MiB memory regions from the operating system and then manages it independently. While this is fine for systems that support demand paging, this strategy increases the minimal space requirement to at least 256 MiB for Unikraft, hence the observation of 261 MiB. Note that this issue could be addressed by providing a new region backend to Mimalloc that does not enforce such memory requirements. However, this requires significant engineering that outreaches the scope of this thesis. We therefore consider this observation as a limitation of Mimalloc for systems that do not support demand paging.

tinyalloc present an initially higher memory usage comparatively to TLSF and BBUDDY due to the preallocation of allocator metadata (fresh blocks) at system startup. Nevertheless, the overall increase of memory between 10 and 100,000 queries remains low (4.4x) compared to TLSF (14.8x).

Finally, TLSF presents the overall lowest memory usage. For the first 1000 queries the memory footprint stays identical whereas tinyalloc already shows an increase of 35%. Nevertheless, the overall fragmentation remains suboptimal for longer workloads, due to TLSF's use of size classes and subsequent rounding.

### 5.5.2 Redis

We determine the minimal amount of memory required to (1) start a functional Unikraft Redis unikernel and (2) successfully perform the benchmark that we realized in Subsection 5.4.2. To this end, we leverage binary search to return the smallest amount of guest memory for which (1) `ping` and (2) `redis-benchmark` execute successfully. Table 5.9 present our results.

Our findings are similar to the previous subsection: we observe a difference of a few mebibytes between tinyalloc and TLSF that are due to the number of fresh blocks initialized. Mimalloc suffers from excess memory usage due to its initial region allocation. Finally, BBUDDY suffers from a pathological internal fragmentation, with an overall x7-9 larger memory footprint compared to TLSF. Overall, allocator specialization allows to reduce the memory consumption by at least 22% (tinyalloc/TLSF).

| | `ping` | `redis-benchmark` |
|---|---|---|
| bbuddy | 61 | 67 |
| mimalloc | 261 | 261 |
| tinyalloc | 9 | 11 |
| tlsf | 7 | 9 |

Table 5.9: Memory usage of Redis Unikraft, with varying allocators and workloads.

### 5.5.3 Nginx

Similarly to the previous subsection, we determine the minimal amount of memory required to (1) start a functional Nginx server and (2) successfully perform the benchmark that we previously realized in Subsection 5.4.3. Table 5.10 presents our results.

Our results are consistent with previous measurements: TLSF reduces the memory footprint by 50% compared to tinyalloc, and 160% compared to BBUDDY. Unlike Redis, we do not observe an increase of memory usage between the workloads which is not surprising since the web server essentially *serves* content (logs are disabled, and we query a single static file).

| | `ping` | `wrk` |
|---|---|---|
| buddy | 13 | 13 |
| mimalloc | 261 | 261 |
| tinyalloc | 10 | 10 |
| tlsf | 5 | 5 |

Table 5.10: Memory usage of Nginx Unikraft, with varying allocators and workloads.

### 5.5.4 Conclusion

Memory footprint shows the clearest differentiation between memory allocators among all KPIs. We show that, by choosing TLSF over other allocators, memory consumption can consistently be reduced by at least 22% for a Redis unikernel and 50% for an Nginx unikernel. Our results also highlight that using unmodified user-space allocators in a system that does not support demand paging can lead to a significant waste of memory. Mimalloc, for instance, consistently requires 261 MiB of memory, regardless of the actual memory usage of the unikernel. Overall, these results are interesting because they show that these is no clear cut; what we find to be optimal for runtime performance (Mimalloc) performs, in fact, significantly worse for runtime memory usage.

# Chapter 6

# Conclusion and Future Work

Unikernels achieve high performance, low memory footprint, and fast boot time with a small trusted computing base. These characteristics are obtained by specializing operating system components. Amongst the classical system components, the dynamic memory allocator is known to have a crucial impact on performance, and its specialization potential is well recognized in the literature. Despite of this, unikernels do not typically specialize the memory allocator.

In this thesis, we explored the idea of leveraging memory allocators as a specialization layer in unikernels. We first provided a systematic analysis of memory allocation in unikernels. We showed that unikernels and general-purpose OSs are subject to similar constraints with regard to memory allocation (e.g., resident memory, physically contiguous memory) but highlighted that, depending on the unikernel implementation, some of these constraints might be implicitly satisfied. We stressed that unikernels do not play by the same rules as general-purpose OSs: system calls are simple function calls and it is common for unikernels to exclusively target uniprocessor systems. As a result, general-purpose allocators that are optimal on general-purpose OSs are not necessarily a best choice in the context of unikernels.

Following our analysis, we implemented allocator specialization in Unikraft, a unikernel framework. We ported Mimalloc, TLSF, and tinyalloc to the Unikraft framework and adapted the existing allocator BBUDDY. We described the technical difficulties that can arise when porting user-space memory allocators to be used as system allocator such as satisfying allocator dependencies at initialization time or providing a uniform API and documented our technical solutions.

Finally, we evaluated our approach in the KVM platform from the perspective of four key performance indicators: (1) image size, (2) boot time, (3) runtime performance, and (4) runtime memory usage. We showed that reasonably small allocators do not impact the image size. On the other hand, allocator specialization can have a significant impact on internal unikernel boot time, and more generally on the total boot time considering specialized VMMs such as Solo5. We found, for instance, that a simple flaw in BBUDDY's initialization function results in an increase of internal boot time of x5.3 for an Nginx unikernel and up to x164 for a helloworld unikernel. We further showed that allocator specialization can bring significant speedups in runtime performance: up to 26% for SQLite with tinyalloc and 18% for Redis with Mimalloc. We highlighted that, considering allocator specialization, not only the application but also the targeted workload should be considered. Finally we evaluated the impact of allocator specialization on runtime memory usage, showing that, by choosing TLSF over other allocators, memory consumption can be reduced by at least 22% for a Redis unikernel and 50% for an Nginx unikernel.

## 6.1 Future Work

In this section, we conclude this thesis with a discussion of the limitations of our approach and draw future lines of research from unexplored paths and ideas.

### Choice of the Allocator

While we demonstrated that there can be significant benefits in specializing the allocator, we stress that these can only be achieved if the allocator is chosen properly. An improper choice could, in fact, lead to a loss of performance; this case was illustrated with tinyalloc in our Nginx measurements. Unfortunately, the process of choosing the *right* allocator remains somewhat of a black art, requiring a certain understanding of the memory allocators, of the application, benchmarking, etc. Further work should address this issue. If the workload is well-defined, this could take the form of an automated system that systematically benchmarks the unikernel with different allocators[1], effectively generalizing and systematizing the manual benchmarks realized in this thesis. Such a system could, for example, be made available to Unikraft's users via the `kraft` toolchain [1].

### Security

Subsection 3.1.2 and 3.1.3 motivated the potential that security-focused allocators can have in unikernels due to the absence of system call costs, which have historically been a limitation. Future research should further study the challenges of providing temporal memory safety in unikernels, and to what extent

[1]Uelgen and Avci [87] explore a similar approach with IMAS in general-purpose OSs.

the characteristics of unikernels (in particular the absence of system call costs) can reduce the costs.

### Per-Component Allocator Specialization

This thesis has focused on specializing the allocator for the unikernel as a whole. However, we showed that using memory pools in the network stack produces a high performance increase (x3.8-x4.8). Based on this observation, further work could push the specialization a step further, specializing the allocator on a per-component basis, potentially trading off image size and boot time for runtime performance, memory usage, or security.

### Post-Link Heap-Layout Optimization

Finally, we note that the approach taken by Savage and Jones [81] with HALO (introduced in Subsection 2.2.3) could be complimentary this work. HALO takes a profile-guided approach to custom memory allocation: the application (e.g., the unikernel) is run with an instrumented allocator to generate profiling data. Clustering algorithms are then applied to the data, determining groups of memory blocks that are used at the same time. Finally, the application is rebuilt and HALO synthetizes a custom allocator that takes advantage from the clustering information to maximize locality of reference (e.g., by co-locating related blocks). This approach is purely complimentary to ours: the synthesized allocator only handles the allocations that can be associated to a cluster; the remaining ones are passed on to a general-purpose allocator. Furthermore, this approach is associated to a significant degree of fragmentation, effectively trading off memory for performance, which emphasizes the complimentary nature of HALO.

# Appendix A

# Benchmark Data

## A.1 SQLite

| # of queries | mimalloc | tinyalloc | tlsf | bbuddy |
|---|---|---|---|---|
| 10 | 425353 | 336276 | 360489 | 384059 |
| 100 | 1419498 | 1325004 | 1396580 | 1547310 |
| 1000 | 11542240 | 11448888 | 12069495 | 12980661 |
| 10000 | 130335589 | 137487409 | 137755463 | 262148137 |
| 60000 | 1456010505 | 1788309033 | 1659447196 | 5774684236 |
| 100000 | 5123929061 | 6268938839 | 5772283102 | n.a. |

Table A.1: Average execution time of an increasing number of SQL `INSERT` queries in SQLite Unikraft, in nanoseconds (ns). This data is graphed relative to Mimalloc in Figure 5.4.

| Allocator | Mean | $\sigma$ |
|---|---|---|
| bbuddy | 67 | 992 |
| mimalloc | 32 | 17 |
| tlsf | 38 | 4 |
| tinyalloc | 104 | 36 |

Table A.2: Mean execution time of `malloc()` with varying allocators under an increasing number of SQL `INSERT` queries, in nanoseconds. The $\sigma$ column provides the standard deviation of our measurements.

# Appendix B

# Reproducibility

## B.1 Network Stack Configuration

Table B.1 presents the optimized network stack configuration used in Chapter 5. Table B.2 presents the network stack (LwIP) configuration used to enable memory pools. We define three custom memory pools of 6000 elements of size 256 bytes, 512 bytes, and 1560 bytes, respectively.

| Parameter | Value | Comments |
|---|---|---|
| `MEMP_NUM_TCP_PCB_LISTEN` | 64 | Max. number of simultaneous TCP listeners |
| `MEMP_NUM_NETCONN` | 64 | Max. number of `struct netconn`, which limits the maximum number of open sockets |
| `MEMP_NUM_TCP_PCB` | 500 | Max. number of simultaneous TCP connections |

Table B.1: LwIP configuration used in Chapter 5.

| Parameter | Value | Comments |
|---|---|---|
| `MEM_LIBC_MALLOC` | 0 | Do not use the allocator from the C standard library |
| `MEM_USE_POOLS` | 1 | Use pool-based memory allocation |
| `MEM_USE_POOLS_TRY_BIGGER_POOLS` | 1 | If adequately sized pools are full, fallback to a bigger pool |
| `MEMP_USE_CUSTOM_POOLS` | 1 | Enable custom memory pools |

Table B.2: Additional LwIP configuration used to enable memory pools.

## B.2 Nginx Configuration

```
worker_processes  1;
daemon off;
master_process off;
access_log off;

events {
  worker_connections 32;
}

http {
  [...]

  keepalive_timeout 65;

  open_file_cache max=200000 inactive=20s;
  open_file_cache_valid 30s;
  open_file_cache_min_uses 2;
  open_file_cache_errors on;

  [...]
}
```

## B.3 Redis Configuration

| Parameter | Value | Comments |
|---|---|---|
| `appendonly` | no | Disable Append-Only File (AOF) |
| `save` | *empty* | Disable Redis Database Backup (RDB) |
| `logfile` | warning | Only log critical events |

Table B.3: Custom Redis configuration used in Chapter 5.

# Bibliography

[1] Unikraft's User Guide — Getting started with kraft. URL `http://docs.unikraft.org/kraft.html`. Online; accessed May 04, 2020.

[2] The Linux Kernel User's and Administrator's Guide v5.7.0-rc2 — The kernel's command-line parameters. URL `https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html`. Online; accessed April 24, 2020.

[3] musl — lightweight, fast, simple, and free C standard library. URL `https://musl.libc.org/`. Online; accessed May 18, 2020.

[4] newlib — a C library intended for use on embedded systems. URL `https://www.sourceware.org/newlib/`. Online; accessed May 18, 2020.

[5] How fast is redis? URL `https://redis.io/topics/benchmarks`. Online; accessed May 18, 2020.

[6] Solo5 — a sandboxed execution environment for unikernels. Online; accessed May 24, 2020.

[7] Unikraft's Developer Guide — Debugging in Unikraft. URL `http://docs.unikraft.org/developers-debugging.html`. Online; accessed May 04, 2020.

[8] wrk – modern http benchmarking tool. URL `https://github.com/wg/wrk`. Online; accessed May 24, 2020.

[9] *IEEE Standard for Information Technology – Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7.* Institute of Electrical and Electronics Engineers (IEEE), January 2018.

[10] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX*

*Symposium on Networked Systems Design and Implementation*, pages 419–434. USENIX Association, February 2020.

[11] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[12] Emery D Berger and Benjamin G Zorn. DieHard: probabilistic memory safety for unsafe languages. In *27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 41, pages 158–168. Association for Computing Machinery, 2006.

[13] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *9th international conference on Architectural support for programming languages and operating systems.* Association for Computing Machinery, 2000.

[14] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Composing high-performance memory allocators. In *ACM SIGPLAN Conference on programming language design and implementation*, volume 1, pages 114–124. Association for Computing Machinery, 2001.

[15] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Reconsidering custom memory allocation. In *17th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications.* Association for Computing Machinery, 2002.

[16] A. Boicea, F. Radulescu, and L. I. Agapin. MongoDB vs Oracle – Database Comparison. In *3rd International Conference on Emerging Intelligent Data and Web Technologies*, pages 330–335, 2012.

[17] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer 1994 Technical Conference.* USENIX Association, June 1994.

[18] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *7th International Conference on Cloud Computing Technology and Science*, pages 250–257, 2015.

[19] Alfred Bratterud, Andreas Happe, and Robert Anderson Keith Duncan. Enhancing Cloud Security and Privacy: The Unikernel Solution. In *8th International Conference on Cloud Computing, GRIDs, and Virtualization*, February 2017.

[20] Josiah L. Carlson. *Redis in Action*, chapter 4, pages 84–87. Manning Publications Co., 2013.

[21] Jonathan Corbet. Virtually mapped kernel stacks. *Linux Weekly News*, June 2016. URL `https://lwn.net/Articles/692208/`.

[22] Jonathan Corbet. The current state of kernel page-table isolation. *Linux Weekly News*, December 2017. URL `https://lwn.net/Articles/741878/`.

[23] Jonathan Corbet, Rubini Alessandro, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*, chapter 8, pages 213–234. O'Reilly Media, February 2009.

[24] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th USENIX Conference on Security Symposium*, pages 815–832, 2017.

[25] L. De Simone and G. Mazzeo. Isolating Real-Time Safety-Critical Embedded Systems via SGX-Based Lightweight Virtualization. In *9th IEEE International Workshop on Software Certification*, pages 308–313, October 2019.

[26] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks*, pages 269–280, July 2006.

[27] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *24th IEEE international conference on advanced information networking and applications*, pages 27–33, 2010.

[28] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *8th USENIX Conference on Operating Systems Design and Implementation*, pages 339–354, 2008.

[29] Rohit Dube. A comparison of the memory management sub-systems in freebsd and linux. Technical report, Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland, September 1998.

[30] Dominik Durner, Viktor Leis, and Thomas Neumann. On the impact of memory allocation on high-performance query processing. In *15th International Workshop on Data Management on New Hardware*. Association for Computing Machinery, July 2019.

[31] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. I/O Is Faster Than the CPU: Let's Partition Resources and Eliminate (Most) OS Abstractions. In *17th Workshop on Hot Topics in Operating Systems*, pages 81–87. Association for Computing Machinery, 2019.

[32] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, December 1995.

[33] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *BSDCan conference*, Ottawa, Canada, April 2006.

[34] *FreeBSD Handbook*. The FreeBSD Documentation Project. URL `https://www.freebsd.org/doc/en/books/handbook`. Online; accessed April 29, 2020.

[35] Stefano Garzarella. How to measure the boot time of a linux vm with qemu/kvm, August 2019. URL `https://stefano-garzarella.github.io/posts/2019-08-24-qemu-linux-boot-time/`. Online; accessed May 18, 2020.

[36] David Gay and Alex Aiken. Memory Management with Explicit Regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323. Association for Computing Machinery, 1998.

[37] Carl Gebhardt, Chris I. Dalton, and Allan Tomlinson. Separating Hypervisor Trusted Computing Base Supported by Hardware. In *5th ACM Workshop on Scalable Trusted Computing*, pages 79–84. Association for Computing Machinery, 2010.

[38] T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, and F. De Turck. Unikernels vs Containers: An In-Depth Benchmarking Study in the Context of Microservice Applications. In *8th International Symposium on Cloud and Service Computing (SC2)*, pages 1–8, November 2018.

[39] Mel Gorman. *Understanding the Linux virtual memory manager*. Bruce Perens' Open Source series. Prentice Hall, Upper Saddle River, NJ, 2004.

[40] Dirk Grunwald and Benjamin Zorn. Customalloc: Efficient synthesized memory allocators. *Software: Practice and Experience*, 23(8):851–869, August 1993.

[41] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 161–176. Springer International Publishing, 2017.

[42] Serge Hallyn and Michael Kerrisk. `CGROUPS(7)`. In *Linux Programmer's Manual*. Linux man-pages project, November 2019. URL `http://man7.org/linux/man-pages/man7/cgroups.7.html`. Release 5.05.

[43] Yusuf Hasan and J. Morris Chang. A tunable hybrid memory allocator. *Journal of Systems and Software*, 79(8):1051–1063, August 2006.

[44] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. Intel Corporation, September 2016. Chapter 17.15.

[45] Alin Jula and Lawrence Rauchwerger. Custom Memory Allocation for Free. In *Languages and Compilers for Parallel Computing*, pages 299–313. Springer, November 2007.

[46] Alin Jula and Lawrence Rauchwerger. Two Memory Allocators That Use Hints to Improve Locality. In *International Symposium on Memory Management*, pages 109–118. Association for Computing Machinery, 2009.

[47] Jun Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems*, pages 260–269, October 2003.

[48] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. Mallacc: Accelerating memory allocation. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 33–45. Association for Computing Machinery, 2017.

[49] Alok Kataria. CPUID usage for interaction between Hypervisors and Linux, October 2008. URL `https://lwn.net/Articles/301888/`. Online; accessed May 06, 2020.

[50] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.

[51] Michael Kerrisk and Eric Biederman. `NAMESPACES(7)`. In *Linux Programmer's Manual*. Linux man-pages project, August 2019. URL `http://man7.org/linux/man-pages/man7/namespaces.7.html`. Release 5.05.

[52] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav HarEl, Don Marti, and Vlad Zolotarov. OSv - Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference*, pages 61–72, 2014.

[53] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968. pp. 435–455.

[54] Joseph Kong. *FreeBSD Device Drivers*, chapter 2, pages 17–25. No Starch Press, May 2012.

[55] Greg Kroah-Hartman. The kernel configuration and build process. *Linux Journal*, 2003(109):3, May 2003.

[56] Simon Kuenzer. Unikraft: Unikernels made easy. In *32nd Large Installation System Administration Conference*, 2018. URL https://www.usenix.org/conference/lisa18/presentation/kuenzer.

[57] Alexey Kukanov and Michael J. Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(04):309–322, November 2007.

[58] Stefan Lankes, Simon Pickartz, and Jens Breitbart. HermitCore: A Unikernel for Extreme Scale Computing. In *6th International Workshop on Runtime and Operating Systems for Supercomputers*. Association for Computing Machinery, 2016.

[59] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring rust for unikernel development. In *10th Workshop on Programming Languages and Operating Systems*, pages 8–15. Association for Computing Machinery, 2019.

[60] Per-Ake Larson and Murali Krishnan. Memory allocation for long-running server applications. In *1st International Symposium on Memory Management*, pages 176–185, 1998.

[61] Sangho Lee, Teresa Johnson, and Easwaran Raman. Feedback Directed Optimization of TCMalloc. In *Workshop on Memory Systems Performance and Correctness*. Association for Computing Machinery, June 2014.

[62] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. In *Programming Languages and Systems*, pages 244–265. Springer International Publishing, 2019.

[63] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. Snmalloc: A message passing allocator. In *International Symposium on Memory Management*, pages 122–135. Association for Computing Machinery, 2019.

[64] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon

Crowcroft. Unikernels: Library operating systems for the cloud. *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 461–472, 2013.

[65] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-In-Time Summoning of Unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation*, pages 559–573. USENIX Association, May 2015.

[66] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *26th Symposium on Operating Systems Principles*, pages 218–233. Association for Computing Machinery, 2017.

[67] M. Mao and M. Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *5th International Conference on Cloud Computing*, pages 423–430, 2012.

[68] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473. USENIX Association, April 2014.

[69] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: a new dynamic memory allocator for real-time systems. In *16th Euromicro Conference on Real-Time Systems*, pages 79–88, 2004.

[70] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46. Association for Computing Machinery, 2004.

[71] T. L. Nguyen and A. Lebre. Virtual Machine Boot Time Model. In *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 430–437, 2017.

[72] Yoshinori K. Okuji, Brian Ford, Erich Stefan Boleyn, and Kunihiro Ishiguro. *Multiboot Specification*. Free Software Foundation, 2019. URL `https://www.gnu.org/software/grub/manual/multiboot/`.

[73] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A Binary-Compatible Unikernel. In *15th ACM SIGPLAN/SIGOPS*

*International Conference on Virtual Execution Environments*, pages 59–73. Association for Computing Machinery, 2019.

[74] Pierre Olivier, Antonio Barbalace, and Binoy Ravindran. The case for intra-unikernel isolation. In *10th Workshop on Systems for Post-Moore Architectures*, April 2020.

[75] SeongJae Park, Minchan Kim, and Heon Y. Yeom. GCMA: Guaranteed Contiguous Memory Allocator. *ACM Special Interest Group on Embedded Systems Review*, 13(1):29–34, March 2016.

[76] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–304. Association for Computing Machinery, 2011.

[77] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An Analysis of Performance Evolution of Linux's Core Operations. In *27th ACM Symposium on Operating Systems Principles*, pages 554–569. Association for Computing Machinery, 2019.

[78] Marc Rittinghaus. System call aggregation for a hybrid thread model. Study thesis, Karlsruhe Institute of Technology, December 2010. URL `https://os.itec.kit.edu/21_2210.php`.

[79] Andreas Rossberg, editor. *WebAssembly Specification, Release 1.1*, chapter 'Instructions'. WebAssembly Community Group, April 2020. URL `https://webassembly.github.io/spec/core/index.html`.

[80] Richard L. Rudell. Multiple-valued logic minimization for pla synthesis. Technical report, Electronics Research Laboratory, University of California, June 1986.

[81] Joe Savage and Timothy M. Jones. HALO: Post-Link Heap-Layout Optimisation. In *18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 94–106. Association for Computing Machinery, 2020.

[82] Karsten Schmidt. thi.ng/tinyalloc: malloc / free replacement for unmanaged, linear memory situations. URL `https://github.com/thi-ng/tinyalloc`. Online; accessed April 20, 2020.

[83] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and Virtual Machines at Scale: A Comparative Study. In *17th International Middleware Conference*. Association for Computing Machinery, 2016.

[84] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Conference on Operating Systems Design and Implementation*, pages 33–46. USENIX Association, 2010.

[85] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 143–156. Association for Computing Machinery, 2020.

[86] Michael S. Tsirkin and Huck Cornelia, editors. *Virtual I/O Device, Version 1.1*, chapter 2 'Basic Facilities of a Virtio Device'. OASIS Open, April 2019. URL `https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html`. Committee Specification 01.

[87] Onur Uelgen and Mutlu Avci. The intelligent memory allocator selector. *Computer Languages, Systems & Structures*, 44:342–354, 2015.

[88] Kiem-Phong Vo. Vmalloc: A General and Efficient Memory Allocator. *Software: Practice and Experience*, 26(3):357–374, March 1996.

[89] Adam Wick. The HaLVM: A Simple Platform for Simple Platforms. *XenSummit*, August 2012. URL `http://www-archive.xenproject.org/xensummit/xs12na_talks/M9b.html`.

[90] Dan Williams and Ricardo Koller. Unikernel Monitors: Extending Minimalism Outside of the Box. In *8th USENIX Conference on Hot Topics in Cloud Computing*, pages 71–76. USENIX Association, 2016.

[91] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, pages 1–116. Springer, 1995.

[92] Heymian Wong. PCI Express Multi-Root Switch Reconfiguration During System Operation. Master's thesis, Massachusetts Institute of Technology, May 2011. URL `https://dspace.mit.edu/handle/1721.1/66819`. Section 3.4 – PCI Bus Enumeration.

[93] B. Xavier, T. Ferreto, and L. Jersak. Time Provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 277–280, 2016.

[94] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. Why nothing matters: The impact of zeroing. In *11th ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 307–324. Association for Computing Machinery, 2011.

[95] Hao Zhang, Bogdan Marius Tudor, Gang Chen, and Beng Chin Ooi. Efficient In-Memory Data Management: An Analysis. *VLDB Endowment Journal*, 7 (10):833–836, June 2014.

[96] Benjamin Zorn and Dirk Grunwald. Empirical Measurements of Six Allocation-Intensive C Programs. *ACM SIGPLAN Notices*, 27(12):71–80, December 1992.