Ultrafork and Focused KSM:

Tools for Memory Optimization in Duplicate Containers

A Major Qualifying Project Report

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

in

Computer Science

by

Adrian Curless

Alexander Simoneau

William Cross

March 21, 2022

APPROVED:

Professor Craig A. Shue, Project Advisor

This report represents work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review.

Abstract

Containers are an increasingly common tool for providing isolation between applications running on the same hardware. Increased isolation provides protection from attacks at the cost of increased memory usage. Common memory sharing techniques such as page sharing and deduplication do not work well across container boundaries. This work presents two tools for sharing memory between containers, Ultrafork and Focused KSM. Ultrafork provides a way to clone containers in a similar manner to how `fork` clones processes. The cloned containers share memory copy-on-write with the original container, preventing needless copying when containers start. Focused KSM provides an implementation of Kernel Same-Page Merging specific to containers. It is able to merge duplicate pages while containers are running. Together, Ultrafork and Focused KSM provide robust and effective memory optimization for containers. We show that significant memory savings can be achieved with only a small CPU overhead.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

One of the security measures that virtual hosting providers and other companies that host cloud services can implement in their data centers is isolation, where each server gives connected users separate containers to ensure that clients cannot access restricted resources. However, this approach comes at the cost of significant memory usage. There is an inherent cost to using containers, as there is base information that each container needs its own copy of, meaning that a server has to store many copies of the same information along with the data that is unique to each container. In large data centers, such as those run by Instructure hosting their Canvas online learning platform, there may be thousands of users connected. In this case, if individual containers were used, the memory used for each container would rapidly add up and put heavy strain on the server. The organization running the data center can solve this by purchasing additional RAM to increase memory space, but the amount needed to support the needs of full container isolation may be prohibitively expensive.

The SuS (Single Use Servers) system is one method for introducing isolation to servers, which can increase security by stopping malicious clients from interacting or using confused deputy attacks to access restricted resources [16]. However, SuS encounters the same problem described above, where the memory cost to implement the system requires hosts to spend far too much on RAM, thus making the system too expensive to be worth using.

Our goal is to improve the SuS memory usage and make it cost-effective to run. We achieve this through a system of deduplication that can eliminate the duplicate information from each container through copy-on-write sharing and page merging.

We propose two solutions to memory deduplication of container processes that both utilize a proactive approach to memory deduplication. Both solutions are specifically designed for the problem of memory sharing across nearly identical containers. Proactive memory deduplication is the deduplication of pages before or immediately after memory is allocated by a container process.

Our first proposed solution operates before a container process is finished forking. Instead of completing a full fork, we instead create a cloned container process using a "pristine" copy of the original container as reference. The pristine copy exists at the state right before a new user loads into their SuS container. This clone is then altered over time by performing Copy-on-Write (COW) operations and continued low-priority deduplication as tasks are performed on the server. Reducing our memory spikes with high volume provisioning that can occur at peak times is a critical design feature of this approach. We will call this enhanced forking approach Ultrafork.

Focused KSM is our second proposed solution operates after a container process starts and memory is being allocated. Instead of allowing KSM to continuously search page tables for memory to deduplicate, we hijack its focus towards the currently running container processes we care about. After a period of time, the priority focus will be dropped and continuous deduplication will continue at a lower priority using idle CPU cycles. This approach is designed to flatten spikes in memory usage that occur when many users load at the same time similar to UltraFork, but as an extension of work done on KSM as opposed to a fully novel solution.

Webhosting providers, who host a large number of servers on the same hardware, often provide minimal isolation. This work provides a way to add isolation (through docker containers), increasing the security offered, while keeping memory costs low.

The UltraFork and Focused KSM techniques together enable memory deduplication for the SuS system, reducing the memory cost of additional Docker containers significantly. This memory reduction is achieved with a minimal increase in CPU usage at runtime, less than 13% in our trials. The techniques presented allow for a larger number of containers to be run on the same physical hardware, maximizing the cost effectiveness of the system.

Large amounts of memory are required to run a system with many duplicated containers. The cost per gigabyte for Dell DDR4 2666MHz ECC RAM for the Power-Edge R6145 server was $7.05 in October of 2021. The tenfold reduction in RAM usage means that more value can be obtained from already purchased RAM. A virtual hosting provider could support 10 times as many containers on their servers as they did before deploying this work. Additionally, in the future, servers with a tenth the RAM could be purchased to replace the servers that were in use before deployment of UltraFork and Focused KSM while supporting the same number of containers with comparable performance.

To implement memory sharing, Focused KSM incurs some CPU overhead. This is the case with all current work in the area of page deduplication. Any page deduplication method has the potential to cause performance issues while page tables are being scanned. With Focused KSM this CPU overhead is minimal and should not be noticeable for users. The benefit of having effective memory sharing far outweighs the cost of the minimal performance penalty.

1.1 Competition

Our main competitor is KSM (Kernel Same-page merging) [2], a page merging system that is already mainlined in the Linux kernel. KSM works by scanning virtual memory areas (VMA) for duplicate pages. Duplicates are identified using a checksumming algorithm. Metadata about pages is stored in two metadata trees for faster access. KSM is a purely reactive approach to page deduplication, and is the only competition that exists in the main line Linux kernel today. KSM requires pages to be explictly marked as "mergeable" in order to be considered for merging.

UKSM (Ultra Kernel Same-page Merging) [31] is a KSM derivative using a hierarchical hash algorithm to improve page scanning performance. The hierarchical hash model allows a weak, fast hash to be used to compare pages that are likely to differ significantly, while a slower, more accurate hash can be used to compare pages with more subtle differences. UKSM achieves significant performance increases over KSM. UKSM does not require explicitly marking pages as mergeable, like KSM, making it more easily deployable.

Our final major competitor is PageCmp [21], which is much like KSM but with a hardware offload. Memory comparisons are done by leveraging the memory controller. This provides improvements in page comparison performance as well as a reduction in power usage.

1.2 Research Questions

We considered the following questions when beginning our work on UltraFork and FKSM:

1. Is it possible to predictively deduplicate memory before allocation occurs?
2. How can deduplicated memory be effectively shared in a way that does not significantly impact performance while also preserving security?
3. To what extent is deduplicating container memory possible?
4. How can the overhead of current memory deduplication techniques be reduced?
5. How can a fork point be effectively chosen to limit duplicated memory after forking?
6. What existing technologies would benefit from this system and how could they be supported?

7. How can the amount of Copy-on-Write memory a process has be measured?

Chapter 2

Related Work

While doing background research, we identified a number of categories that most related work can be sorted into. Much work has been done to optimize memory usage for Virtual Machines. Page sharing and deduplication is another major area of research. Containers and serverless computing have become increasingly popular in industry, and research has been done into memory efficiency in this area as well. Finally, we found articles that discussed using patterns to predictively optimize memory.

2.1 Fork

The `fork()` system call has a long history in Unix operating systems. The call is used to create a new process by duplicating the address space of the caller process. Originally, this was done by physically copying the address space of the caller process into a new section of memory for the newly spawned process to use. This approach was clearly wasteful, both in terms of CPU time and memory usage, especially when considering the case when the two processes have largely the same memory content. The `fork()` call was enhanced to be copy-on-write. This means that the address space of the caller process was no longer duplicated, each process contained references to the same pages of memory. When one process needed to write to a memory page, that page would be copied and the write would occur in the newly allocated un-shared page. This preserves virtual memory isolation between processes while improving upon the performance and resource usage of the `fork()` system call [23]. In Section 4.3.2 we evaluate the performance of `fork` for varing numbers of processes and compare it to the performance of Ultrafork.

2.2 Virtual Machines

Xen is an open source type 1 hypervisor commonly used for virtualization. A type 1 hypervisor runs on the hardware directly, without an operating system between. Xen aims to provide low cost virtualization for Linux and other platforms. It can emulate a variety of CPU architectures, including x86-64 and arm. Xen development has a particular focus on security and making virtual machines cost effective in terms of resource usage [5].

Potemkin [27] is a system based on the Xen hypervisor for efficient creation of honey pot virtual machines with memory sharing. The authors introduce flash cloning, a technique for starting a lightweight virtual machine from a reference image. The reference image is kept separate from the honey pot VMs and does not respond to requests. It solely acts as a defined point to start the flash clone from. Copy-on-Write is implemented for the cloned virtual machine through write-protected pages in the newly created Xen domain. The system involves modifications to the Xen hypervisor.

Our implementation differs from Potemkin as Potemkin is a modification to Xen, and implements its own lightweight container system. Our UltraFork approach, while similar, is designed to work with unmodified Docker containers, building on the already well understood and extensively reviewed Docker containerization process and security model.

SnowFlock [15] is a system built for distributed computing that allows rapid creation of identical virtual machines on separate hosts. It implements a version of the standard fork() system call that works for virtual machines rather than processes, accomplished using modifications to the Xen virtual machine manager and a suite of daemons running in domain() that control the cloning and deallocation of VMs. When it attempts to fork a VM, SnowFlock first creates a "VM Descriptor", a small file that contains the metadata of the VM and information about the guest kernel, and then fetches states from the parent to populate the new clone's memory so that it can continue execution. SnowFlock also employs a plug-in architecture to allow it to interface with cluster management software; once the clones are successfully created, SnowFlock can defer to the cluster management software it is currently connected with to determine which physical hosts the new child VMs will be allocated to.

UltraFork differs from SnowFlock [15] in exactly what it saves in its pristine copy and how new VMs interact with that copy. SnowFlock is based more around replicating the standard fork() system call, allowing a VM to duplicate itself into a number of children. Instead of cloning from an existing VM, UltraFork saves its pristine copy separately, and newly started VMs can replicate that copy directly. SnowFlock's method of creating a VM descriptor file to assist with the cloning process is somewhat similar to UltraFork's pristine copy system, but that VM descriptor only saves the current state of the cloned VM while UltraFork saves a constant pristine copy that can be accessed at any time. Finally, SnowFlock focuses more heavily on distributed computing with its cluster management integration, while UltraFork is designed to work on one single host at a time.

2.3 Page Deduplication and Sharing

KSM (Kernel Same-page Merging) [2] is a page sharing technique implemented in Linux. KSM traverses page tables during time when the CPU is idle to search for duplicate pages. KSM only deduplicates anonymous pages, which are pages that are not backed by files. An anonymous page is typically created as a result of memory allocation in the program [17, p. 318]. KSM relies on pages being explicitly marked with the `madvise` system call. KSM is of limited use in most systems because of the explicit marking requirement and the CPU cost of traversing and hashing many pages.

UKSM [31] is an improvement of KSM that supports automatic scanning of system memory, as opposed to using explicit markings with `madvise` like in KSM. USKM further optimizes on KSM by designating “rich areas” where abundant deduplication is occurring and will prioritize those regions. Zero pages are also considered separately and merged into one unswappable page. Other optimizations include an improvement to the hashing algorithm, volatile page avoidance, and optimized memcmp.

PageCmp [21] is a KSM-based approach to page sharing with a hardware offload for page comparisons. PageCmp aims to improve the performance of KSM by offloading the byte-by-byte page comparisons to the local comparator that exists in DRAM modules. PageCmp advertises a 4x memory bandwidth reduction at the cost of less than 1% added hardware overhead.

Another performance improvement to KSM can be achieved with the use of array mapped tries [32]. In AMT-KSM (Array Mapped Trie KSM), each page is divided into segments, each with their own hashes. The hashes of the segments are concatenated and used as keys for the trie. This reduces search time for duplicated pages when compared with KSM. AMT-KSM also reduced CPU usage by up to 44.9% and memory bandwidth usage by up to 31.6%.

Interdomain Shareable Pages [14] describes the implementation of a driver to perform page analysis for Xen. The authors created a system that allows pages to be shared between virtual machines and with the host. Pages are compared using a fast, non-cryptographic hash algorithm, and then with a byte-by-byte comparison to eliminate the risk of collisions.

One issue stemming from deduplication is a side-channel attack where memory access latency is able to identify running programs in other guest VMs. The technique is described in the work of K. Suzaki et. al. [25]. In SEMMA (Secure Efficient Memory Management Approach in Virtual Environment) [8], a group based secure page sharing approach (GSKSM) is proposed an implemented, eliminating inter-VM sharing of pages. The approach further extended the concept of efficient memory management by implementing memory ballooning for the secure groups.

The ShMVM system [10] is an older system for memory sharing that allows separate Java Virtual Machine processes to share objects while still accessing their own private objects. It divides each processes address space into a private, shared, and indirect space, with the shared and indirect spaces being located at the same virtual addresses for all processes. Objects in a processes' private space can access each other, and any process can access objects in the shared space. However, objects in the shared space cannot directly access private objects since the location in virtual memory of the private space might vary from process to process. As a result, any time a process needs to use a shared object to access one of its own private objects, it makes an entry in its unique "indirection table" that indicates where the reference in that shared object is pointing to. Each process has its own indirection table, which allows the processes to share communal copies of objects but still have those objects reference the processes' own private objects.

The Focused KSM approach presented differs from the related KSM techniques [2], [31], [21] and [32] in that it is specific to the exact use case of the SuS system. Focused KSM only considers pages allocated by the SuS Docker containers for merging. This reduces the number of pages FKSM needs to consider, reducing the load on the CPU incurred by other page scanning work.

2.4 Containers

Docker is a common open source container platform. It allows the creation of lightweight containers which can be used in the deployment and development of applications [7]. Docker containers are managed by a privileged process running on the host known as the Docker daemon. From a security perspective, containers do not provide as much isolation as virtual machines [7], but have a much lower resource overhead, making them a good choice for a system that needs isolation, but cannot afford the cost of large amounts of RAM.

Docker containers provide isolation through a number of Linux kernel APIs. By default, each container runs in its own process namespace, IPC namespace, network and filesystem namespace, restricting the ability of processes running inside the container to communicate with host processes or with processes running inside other containers [7]. Docker can also be used to set limits on resources used by containers, through Linux's cgroups.

Linux Security Modules (such as SELinux, AppArmor, IMA and others) are not integrated with containers, leading to sub-optimal configurations. As a result, Security Namespaces were proposed as a way to configure LSMs for use inside containers [24]. The security namespaces were implemented using a modified version of the `clone` system call. A similar approach could be used in this work for the UltraFork mechanism.

Shimmy [1] describes a technique for sharing memory between containers. Their implementation relies on a process running on the host that allocates shared memory segments, which are shared into containers by joining the container and host in an IPC namespace. Shimmy allows memory to be shared between containers on the same machine as well as in a container orchestration system such as Kubernetes.

Reg [29] is a containerization and memory sharing system that maximizes memory sharing by extracting library and binary files that would normally be duplicated across multiple containers into a single layer. Reg also makes its containers more lightweight by making custom runtime environments for each container. Instead of giving each container a generic environment with common programs that may not be used in production, Reg allows developers to define a configuration file that indicates what they need in their container's environment, and builds custom environments based on those config files.

X-Containers [22] presents a new approach to containerization. They propose the use of a LibOS (sometimes called an 'application kernel') as a way to reduce the overhead of system and hyper calls. A specially compiled, minimal Linux kernel is linked as library inside each container. This increases memory overhead, but reduces time spent in the kernel executing code for system and hypercalls. The system also proposes a way to share pages of the LibOS between containers, which provides some memory savings. The presented UltraFork technique is designed for Docker containers, and does not require modifications to the container runtime like X-Containers do. X-Containers have a significant overhead associated with the replication of the kernel-space as a userspace library. Even when deduplicated, this overhead is significant when compared to UltraFork.

2.5 Serverless Computing

Catalyzer [11] is a system for fast initialization of serverless systems. It uses a fork-based approach to quickly start services to respond to user requests. The Catalyzer system is built on top of gVisor [4], an application kernel for containers written in Go. Their approach involves the implementation of a new primitive called sfork (Sandbox Fork), which clones a sandbox from a predefined 'shared' point. The cloning process works with multithreaded programs through modifications to the golang runtime. Catalyzer consists of modifications to gVisor, the golang runtime and the Linux kernel. It is not open sourced.

Catalyzer contains significant modifications to the userspace side of the container system. It is implemented through modifications to Golang itself, the container runtime and gVisor. UltraFork does not modify Docker, and does not require a modified language runtime to operate. This makes it more generic and more easily deployable in the field. The sandbox fork approach described in Catalyzer is similar to UltraFork, but does not include a mechanism for eliminating duplication between containers after they are started.

vHive [26] is a framework for experimentation with serverless systems that is used for analyzing their performance. The developers of vHive used the framework to analyze the practice of snapshotting used in various serverless systems like the aforementioned Catalyzer, and found that a major problem with the serverless model is the "cold-start" delay. In order to save memory, serverless systems usually remove idle instances of functions if they stay inactive long enough. This means that starting a function after a period of inactivity leads to a much larger delay, since it needs to fully re-initialize. Even with snapshotting in place to reduce the initial delay, the developers of vHive found that there was still a much larger delay starting a function from a snapshot than using an existing running instance. To assist with this, the developers then created REAP (REcord And Prefectch) [26]. REAP records a working set file for a function the first time it runs that contains a copy of all the guest memory pages it accessed and a trace file that contains their offsets. In the future, the working set file can rapidly be brought into guest memory when the function is initialized, reducing load time.

While REAP's methodology is similar to the concept of UltraFork, it differs in what specific structures it works with. REAP is designed for use with serverless architecures, and thus saves the working set and guest memory for a specific function in the serverless system. UltraFork is designed for SuS and stores common memory for containers running in SuS. Additionally, REAP is more focused on improving startup time when initializing functions, and does not contain systems for memory sharing, unlike UltraFork which is focused more specifically on reducing memory load by allowing VMs to work off of a pristine copy that contains common memory elements.

Replayable Execution [28] can be leveraged to improve the responsiveness of FaaS (Function as a Service) systems. Replayable execution relies on mapping a checkpoint created from a process as a memory mapped file. This allows for memory to be shared between processes without the overhead of a system like KSM. The authors implement a system where memory is shared between JVMs using this technique to reduce the startup time, which is a significant bottleneck for JVM language FaaS platforms. The implementation of replayable execution is the Checkpoint and Restore in Userspace (CRIU) system, which was implemented for Linux. The CRIU system can be modified to work in Docker containers, but requires breaking the isolation model and running without a SECCOMP filter. UltraFork accomplishes similar memory savings, but without the reduction in security.

Additional work was done by Oakes et al. [20] to develop serverless optimized containers (SOCK) that leverage a Zygote provisioning strategy among other improvements. In Android, application processes are created from a Zygote which serves as the nucleus that forks off new processes. SOCK implements a system where a process is forked but is put in a new container using this Zygote spawning paradigm. Each container runs a special helper process that is used to facilitate data data transfer between the host and the container. Various Zygotes are maintained with different imported libraries to spawn off a variety of containers with different library requirements. It reduces memory and CPU requirements for serverless infrastructure by having a variety of libraries pre-initialized and loaded in memory.

UltraFork is distinct from SOCK [20] in that it maintains a single pristine copy, whereas SOCK organizes a number of Zygotes into a tree structure. When a new container is started in SOCK, its dependencies are analyzed, and the Zygote with the most similar dependency structure is used as a basis for the new container. The technical details of starting a new container from a Zygote and forking a container off the pristine copy in UltraFork are similar, but the problem domain is different. UltraFork uses a single pristine copy, which is a common basis for all containers that will be started. SOCK starts a large number of similar containers, and selects which one is closest to the target environment for the new container. If a container with the exact dependency structure does not exist, initialization will continue in the newly started container, before the application code is executed. UltraFork relies on a known common point for all containers, so this extra step of initialization will never be necessary. In addition, Ultrafork and Focused KSM are designed to use standard API and manually exposed kernel symbols, rather than custom APIs.

2.6 Predictive resource management

Resource management in cloud providers utilizes techniques such as memory ballooning and working set size modeling to allocate hardware resources dynamically. Memory ballooning [5] is a technique by which a system can increase or decrease the "balloon" of extra memory allocated to a given virtual machine. Working set size (WSS) modeling [30] involves estimating memory requirements based on performance needs and predictively increasing allocated resources to a machine.

Satori [18] is a system that employs similar page sharing and deduplication as seen in KSM, as well as the memory ballooning method mentioned above, but with an added system that limits the amount of reclaimed memory a VM can use based on how much memory it shares, with VMs that share more memory being allowed to reap greater benefits. This is meant to give VMs an incentive to share memory, and prevents a single VM from monopolizing the reclaimed memory.

Difference Engine [12] attempts to expand on normal page sharing by compressing and deduplicating infrequently accessed pages by tracking the time since a page has been modified and deduplicates sufficiently unused ones. Additionally, Difference Engine increases memory savings by deduplicating `similar` pages, not just identical ones. To do so, Difference Engine creates "patches" that represent a page relative to a reference page, and merges the shared portions of pages that are sufficiently similar to each other.

Ultrafork and Focused KSM are not strictly predictive approaches. Since they are domain-specific to the SuS system, they have more knowledge about how and when memory is being allocated, and can cooperatively work to maximize memory savings, but neither approach actually predicts future memory usage or new container startups.

Chapter 3

Design and Implementation

A kernel module is a program written to execute in kernelspace, and can be loaded into a running kernel dynamically. Kernel modules are a major method of adding functionality to the Linux kernel. Kernel modules have a number of techniques available to communicate with userspace programs, such as through system calls, `ioctls` (IO Controls) or entries in the procfs or sysfs. In this work, we use `ioctls` to communicate with userspace because of their flexibility and ease of use. A system call perhaps the most common method of communication with userpace, but are difficult to add to the kernel source. System calls are defined in architecture specific tables and require patches to the kernel source [17, p. 79]. `ioctls` are another method to execute kernelspace functions for userpace, but are much more flexible. `ioctls` can be defined in modules and do not require architecture specific code.

We implemented the kernelspace portion of our work as a kernel module, which interfaced with userspace using a number of `ioctls`. We made some modifications to the kernel source directly, which are described in the relevant sections below. We made these modifications on the Linux kernel version 5.17, but they should be easily rebased to other versions as required by future work.

Kernel modules do not have the power to call any function in the kernel. The Linux kernel makes select functions available to be called from modules using the `EXPORT_SYMBOL` and `EXPORT_SYMBOL_GPL` macros. In this work, a number of functions are exported to our module, which are not normally exported by an unmodified Linux kernel.

3.1 Ultrafork

The current implementation of Ultrafork works on processes and their descendants. A future version will work on containers directly. Ultrafork implements a modified version of the `clone` call that is made available to userspace through an `ioctl`. The caller provides the process

ID of the parent process that should be cloned. The given process ID and all descendants will be cloned during the Ultrafork operation. Ultrafork is implemented in three phases. The first phase is called the collection phase. It is described in Section 3.1.5. The second phase is called the clone phase and is described in Section 3.1.6. In the traditional `clone` operation, the resulting process is a child of the original process. Since the goal of Ultrafork is to take a tree of processes and produce an identical tree, fork's default behavior is not desirable. Instead, we need to create another tree of processes that is a sibling of the original tree. Phase 3 performs a re-parenting operation that moves the newly forked processes so they are siblings of the original processes. Section 3.1.7 describes this procedure in detail.

The Ultrafork `ioctl` call's arguments are given in Listing 3.1. The argument `pid` is the process ID of the top-most process being forked. This process and all of its children will be forked.

Listing 3.1: Ultrafork argument structure.

```
struct ufrk_ctx {
    unsigned long pid;
};
```

3.1.1 Background on Linux's `clone`

Before describing Ultrafork in detail, We discuss Linux's `clone` call, with particular focus on `fork`, which as a stated previously is a special case of `clone`. In order to clone a process, the system performs the following basic steps. Note that this is a simplification of the actual `clone` process, which is much more complex. The important aspects that are relevant for Ultrafork are listed here, as well as significant steps that should help the reader understand the complexities involved.

1. Flags are checked for inconsistencies
2. The current task structure is duplicated
3. The scheduler is informed about the fork, this includes locking the process being forked.
4. Open file descriptors and other resources are copied into the child task structure
5. The current process's memory mapping is made available to the child process.
6. Namespaces are copied

7. Information about the current process's calling thread of execution is copied into a newly created thread for the child process. Note that if a multithreaded process is forked, only the thread that called the actual `fork` call is copied. Any other threads are ignored.

8. The child process's `sibling` and `parent` references are adjusted.

9. Locks are released and cleanup is done

10. The parent returns to userspace, the child wakes and begins to execute.

3.1.2 The `task_struct` Structure

In Linux, the `task_struct` is the data structure representing a process in the kernel. Every `task_struct` is a member of a single doubly-linked list of tasks that exist on a given system. Each task contains pointers to its parent, children and siblings, allowing for efficient traversal of process hierarchies. Some relevant fields from the task structure are shown in Listing 3.2.

Listing 3.2: Selected fields from the Linux `struct task_struct`. The task structure can be found in `include/linux/sched.h`.

```
struct task_struct {
    struct mm_struct* mm;
    pid_t pid;
    pid_t tgid;
    struct task_struct* parent;
    struct list_head children;
    struct list_head sibling;
};
```

As shown, the task structure contains two fields with type `pid_t`, which is the type of a process identifier. The field `pid` is the process ID, if the task is a process, and the thread ID if the task is a thread. If the task is a process, the `tgid` will be the same as the `pid`; however, if the task is a thread, the `tgid` will be the process ID of the process that owns the thread.

3.1.3 Background on Threads in Linux

Historically, threads have been implemented in a number of different ways in Linux, beginning with a purely userspace implementation. Today, threads are implemented in the kernel, for maximum performance and flexibility. In Linux, a thread is a lightweight process (LWP), which has its own `task_struct`. The difference between threads and processes in kernelspace is dictated by the flags passed to the system call `clone`. Technically, a thread is simply a process that shares more with its parent than other processes do. Threads share virtual memory areas, signal handling, credentials, file descriptors, file system information and SystemV semaphores [6, pp. 29–30]. Threads each have a unique identifier, which is called a thread ID in userspace, but a Process ID in kernelspace [6, pp. 14–15]. This is a common area of confusion. Tasks also have a thread group id (tgid), which represents the process ID in userspace. To clarify, the system call `gettid` returns the task structure's `pid` member, and `getpid` returns the task structure's `tgid` member. In a process's primary thread, the `tgid` and the `pid` are equal.

3.1.4 Preliminary Experiments

To assess the viability of Ultrafork, we performed several experiments to model various aspects of the system. First, we did an experiment to verify our understanding of how memory is made available to the child process after a traditional `fork`. Next, we illustrated how threads and processes can be traversed in kernelspace. Both experiments provided code that we used in the main Ultrafork implementation.

Anonymous VMA Experiment

Before describing this experiment, we will provide some background on how memory is represented in the Linux kernel. In particular, we will describe the basic principles of anonymous memory allocation in Linux to the extent required to understand this work. We note that memory management is a complex topic that will not be fully described here.

Virtual memory in Linux is represented using a structure called `vm_area_struct`. The structure contains a start and end address and some other fields, and is shown in Listing 3.3. When discussing virtual memory, we must differentiate between file backed and anonymous memory. File backed memory is memory that was originally read in the from a file on the filesystem, for example, the code section of an executable program, or the contents of a file. Anonymous memory is memory that is not backed by a file, for example, the runtime variables used by a program. We will focus on anonymous memory in this work [17, p. 318].

Listing 3.3: Excerpt from the `vm_area_struct`, located in `include/linux/mm_types.h`

```
struct vma_area_struct {
    unsigned long vm_start;
```

```
    unsigned long vm_end;
    struct vm_area_struct* vm_next;
    struct vm_area_struct* vm_prev;
    struct mm_struct* vm_mm;
    struct list_head anon_vma_chain;
    struct anon_vma* anon_vma;
};
```

Anonymous memory is represented for a process with a linked list of virtual memory areas called the `anon_vma`, or anonymous virtual memory area. The `anon_vma` makes it easy to access `vm_area_struct` instances for a process. When a process is forked, memory is shared between the parent and the child using the `anon_vma_chain` which acts as a list of `anon_vma`, which, when traversed, allows the child access to all the parent's anonymous memory. In order to make this access copy on write (COW), the virtual memory operations structure (`vm_operations_struct`) is used to hook writes to the virtual memory areas and copy them for writing in the child process [6, pp. 214–215].

Listing 3.4: An excerpt from the `anon_vma_chain` structure. This structure is defined in `include/linux/rmmap.h`

```
struct anon_vma_chain {
    struct vm_area_struct* vma;
    struct anon_vma* anon_vma;
    struct list_head same_vma;
};
```

Now that we have described the basics of virtual memory handling for `fork`, the experiment can be discussed. The objective of this experiment was to traverse the `anon_vma_chain` of a recently forked process to demonstrate that memory from the parent had been made available to the child.

The initial kernelspace portion of this experiment was implemented as an `ioctl`; however, we then changed it to a kprobe. The reason for this is simple: the `anon_vma_chain` is only available to a process when it is actively scheduled, meaning that by the time our `ioctl` executed, the process was in a waiting state, sleeping. As a simple solution, we converted the `ioctl` to a kprobe, which was attached to the `anon_vma_fork` call, which is a function executed during the fork to make the parent's memory accessible to the child after the fork is complete. By attaching the kprobe to this function, we could examine the `anon_vma_chain` before the function returned. This experiment proved to be a success in illustrating how memory is made available to the child process after a fork.

Thread and Process Traversal Experiment

An important difference between Ultrafork and traditional fork is the handling of threads. The traditional fork implementation ignores any threads other than the thread that called `fork()`. The caller is responsible for ensuring consistency when forking a multithreaded process. This approach is not sufficient for Ultrafork. Since containers may have processes with multiple threads, Ultrafork needs to be able to replicate the threads, just like it would replicate processes.

The goal of this experiment was to devise a way to iterate through all the thread and process children of a given process. First, we created a userspace test program. The program simply forked itself, and in each process, started a thread. In each of the 4 threads, `sleep` was called with a long timeout to allow for the kernelspace portion to run.

The kernelspace portion was again implemented as a kprobe, this time attached to the `kernel_clone` function, which is the main function that performs the `clone` and `fork` operations. At the return of this function, the kprobe is executed, and the experiment loops through the process, its child processes and any threads associated with them.

To loop through child processes, the `children` list of the task struct is traversed using the standard Linux list API. Then, for every process, the `for_each_thread` macro is used to iterate through all the threads related to the given process, as shown in Listing 3.5. As a demonstration of the effectiveness of the test, the PID and TGID are printed for each thread and process. This experiment proved to be effective, and provided reusable code for Ultrafork's implementation.

Listing 3.5: Code segment for iterating through each thread in a process.

```
for_each_thread(process, current_thread) {
    pr_info("Visiting thread %d of process %d\n", current_thread->pid,
        process->tgid);
}
```

3.1.5 Collection Phase

The purpose of the collection phase is to determine what processes needs to be cloned. Ultrafork accepts a process ID as an argument, which is looked up and added to a linked list along with all its descendants. In the future, threads will also be added to the list and any metadata required for SysV construct cloning will be collected. After successful completion of the collection phase, Ultrafork moves on to the next phase, the clone phase.

3.1.6 Clone Phase

At the beginning of the clone phase, we traverse the linked list created in the collection phase, and each process is stopped. This is essential to avoid race conditions with userspace communication with the target processes during the clone procedure. Processes are locked later in the traditional `clone` operation but we perform the locking eagerly since we are operating on a number of processes.

In Linux, the `clone` system call is responsible for creating new processes and threads. The `fork` call is a special case of `clone`. The `clone` call creates a copy of process that executed the system call, with modifications made according to the arguments given. The fundamental assumption that `clone` is targeting the caller process is not true with Ultrafork. The Ultrafork operation can be performed on a process from any process (including that process itself, making it equivalent to `clone`). The Linux implementation of `clone` is mainly located in `kernel/fork.c`, which is the site of most kernel modifications made for Ultrafork.

The main function used for `clone` in kernel-space is called `kernel_clone`. It parses the clone flags and populates an internal metadata structure for later use. Then, it calls a helper `copy_process` to do the real work. The `copy_process` function creates a new process that is a copy of the caller. This involves copying open file descriptors, register values and the process environment. The function does not start the new process. For Ultrafork, we implemented custom versions of both the `kernel_clone` and `copy_process`.

The main modification we made to both the `kernel_clone` and `copy_process` functions was to remove the assumption that the calling process should be cloned. Instead, our modified functions pass a `task_struct` pointer to the parent task. We also made this modification to a number of helper calls used by `copy_process`. We had to export the `copy_process` function so it was visible to our module, along with many helper calls.

Additionally, we relaxed some restrictions that exist in the traditional `clone` implementation. For example, Linux's `clone` does not allow the cloning of a process with signals that have not yet been handled. Ultrafork allows this. If a process with pending signals is forked with Ultrafork, the signals will be handled after the process has been woken up, after the entire process structure has been cloned. There is a small chance this will waste computation resources, if for example the process was sent SIGKILL during the fork, but the probability of this is small.

Another change to the standard behavior of `clone` is the removal of fork bomb protection.

Fork bombing is an attack where a process is spawned that recursively forks itself and overloads the system with new processes. Ultrafork is designed to work on containers, so this type of attack does not apply. Fork bombs will be mitigated by the standard Linux protection on the host and within each container, so the only possibility would be a fork bomb caused by recursively spawning containers, which is not possible due to limitations set by the docker daemon. Since the fork bomb is not possible with Ultrafork, protection against it is not necessary. The only potential for abuse is a rogue process calling the `ioctl`, which is mitigated by requiring that callers be privileged users.

After the each process is cloned, Ultrafork performs a few operations on the cloned process in preparation for the re-parenting phase to minimize the amount of metadata that needs to be tracked between phases (and therefore minimize the memory overhead of Ultrafork itself). After a process is cloned by Ultrafork, we adjust its `parent` and `real_parent` pointers to reference the cloned copy of the process' parent. This is necessary for consistency in the process hierarchy whichw e will explain in more detail in a following section. Also, we make a new entry in the PID translation table. The translation table is a table used by Ultrafork during the re-parenting phase. The table contains an entry for each task being cloned. The entry contains both the PID of the process being cloned and the PID of the cloned process. The following section will describe the usage of this table.

Normally, a process would be woken up after the completion of the clone procedure. In the case of Ultrafork, the wake up is deferred until after the re-parenting phase, since the processes are not usable until they have been re-parented.

3.1.7 Re-Parenting Phase

In the re-parenting phase, we move the newly forked processes to be siblings of the original processes. The top-most newly forked process share a parent with the top-most original process, and the new and original trees will have identical structure. To illustrate this, an example is described.

Figure 3.1 shows a parent process and two children before any Ultrafork operation occurs. After the Ultrafork clone operation, the process structure is shown in Figure 3.2. Then, the re-parent operation is executed, and the final resulting process structure is shown in Figure 3.3. This is the process structure when Ultrafork returns to the caller.

To implement re-parenting, we must keep track of what process in the original group corresponds to what newly cloned process. Without this information, newly cloned processes would be treated like members of the original process group, since before the re-parenting phase both the original processes and the newly created processes are part of the same hierarchy. To track this information, we maintaince the PID translation table. During the clone phase, we add an entry to the table for each process that was cloned. The entry consists of two elements, the original process ID and the new process ID (by process ID, we mean the `task_struct`'s `pid` field, which corresponds to a globally unique identifier for a task). The re-parenting phase iterates through each entry of this table and adding the cloned process to its parent's `children` list. Recall that a cloned process' new parent was set in the clone phase in preparation for this operation.

At the end of the re-parenting phase, Ultrafork performs cleanup operations and wakes up the processes. More specifically, the linked list of processes to fork is cleared, and memory released. Then, processes wake up starting with the topmost process (the same order in which they were locked). Then, the call returns to userspace and both groups of processes can execute.

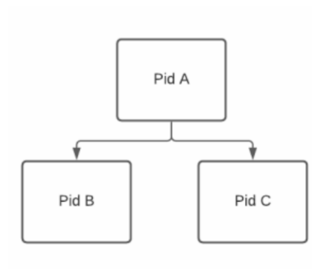

Figure 3.1: A sample process sub-tree before an Ultrafork operation. Processes B and C are children of Process A.

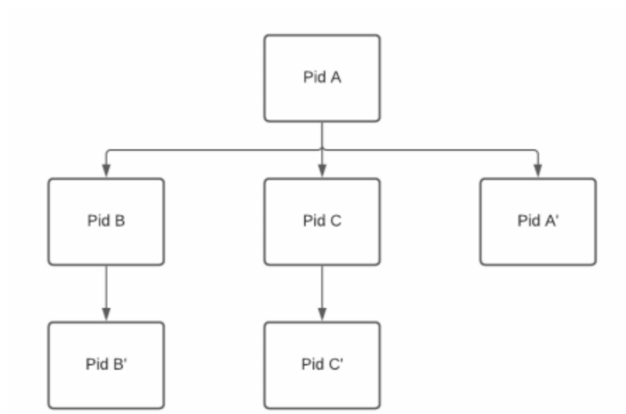

Figure 3.2: The sample process tree after the Ultrafork clone operation but before the re-parent operation. The newly cloned PIDs are A', B' and C' and each is a child of the original process it was cloned from.

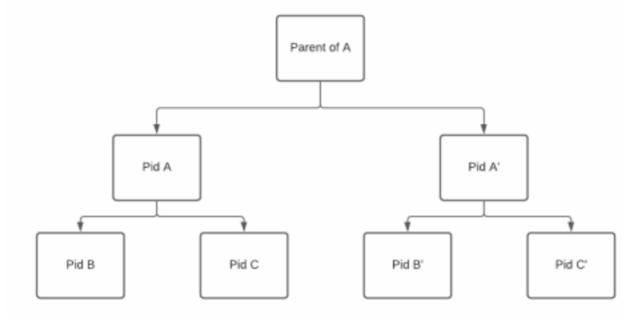

Figure 3.3: The sample process tree as it appears when returned to the caller. Processes A and A' are identical, and are both children of Process A's original parent. Processes B' and C' are identical to processes B and C respectively.

3.1.8 Ultrafork Limitations

The `clone` system call takes a number of flags to specify what aspects of the parent process are shared in the child process. The difference between starting a thread and starting a process with `clone` is the combination of flags passed. Ultrafork uses these flags to control whether a thread or a process is cloned by its internal `copy_process` implementation. This means that most flags are not available for specification by a caller. Future work could expose a subset of the allowed flags and add verification code to ensure the user specified flags are merged with the hard-coded flags in a valid configuration.

The majority of the code for Ultrafork is architecture agnostic except the `copy_thread` function, which is part of the `fork()` process. This function is architecture specific and is modified for Ultrafork. The only supported architecture is x86_64, but other architectures could be added by implementing a modified `copy_thread` function. The function assumes that the calling process (`current`) is the parent process whose thread of execution should be copied. This assumption does not hold true for Ultrafork, so another function was added that passes the parent's task structure as an argument. We also modified several helper functions in the x86_64 implementation.

At various times, Ultrafork performs a lookup on the process ID to get the corresponding `task_struct`. This lookup creates an issue with namespace support. A PID namespace is an isolated environment with unique PIDs inside the namespace, but PIDs are not necessarily unique across namespaces [6, p. 41]. By default, all processes are part of a global namespace, which is what Ultrafork assumes. If Ultrafork is called on a process ID from the non-global namespace, it will not be able to properly look up the process, and could cause a kernel crash. Ultrafork can operate on such processes provided that the global PID is used instead. As an example, consider the topmost process running in a Docker container. Inside the container, that process will have PID 1. Ultrafork cannot simply execute on PID 1, since from outside that container, PID 1 refers to the init process of the system, not the container's process. Ultrafork would instead be called on the globally unique PID for the container process, which is the PID that the process appears to have from outside the container. This can be obtained using tools such as `ps`.

Docker makes use of PID namespaces internally as part of its isolation model. This does not pose an issue for Ultrafork as the namespaces are established after the container processes are started, meaning that at the time Ultrafork executes, all relevant processes are still included in the root namespace.

3.1.9 Focused KSM Instrumentation

Ultrafork can be leveraged to target Focused KSM to avoid unnecessary page scanning. Before Ultrafork returns to userspace, it signals the FKSM module to begin scanning pages of the provided process trees. Then, Focused KSM adds those process trees to its active list, and will consider them for scanning the next time it runs. This is a significant improvement over the naive approach of continuously scanning all pages on a system. With the information provided by Ultrafork, Focused KSM can direct its efforts only to pages in scope that are likely to have been modified. Additionally, tuning is possible with a configurable hang time before the process tree sent from Ultrafork to Focused KSM can be considered for scanning. This allows Focused KSM to ignore the process tree for a time immediately after it has started (and would not differ much from the original process tree it was forked from). This avoids unnecessary page scanning at the beginning of the new process's life cycle.

As an example, consider Processes A and B in Figure 3.4. Process A has recently been Ultraforked into Process B. They both share memory Region 1, which previously existed in Process A and is now available copy on write to Process B.

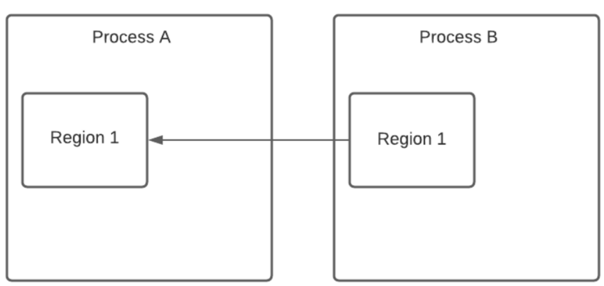

Figure 3.4: An Ultraforked process (A) and its new sibling, B, who share memory region 1.

At some point later in execution of Processes A and B, both processes end up allocating additional identical pages, in Region 2. Figure 3.5 shows the new state. Note that unlike Region 1, Region 2 is not shared, but duplicated in each process.

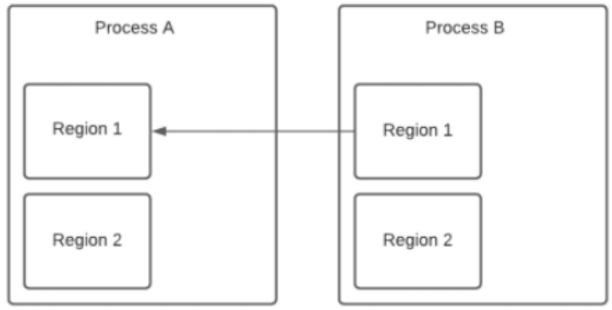

Figure 3.5: The same processes as Figure 3.4, but now with a duplicated memory region, Region 2.

On its own, Ultrafork is unable to mitigate the memory duplication demonstrated in Figure 3.5. However, Focused KSM can handle this situation. Since Focused KSM was informed about the Ultrafork that occurred on Process A to produce Process B, it will scan the pages of both processes, and find the duplicated Region 2. Then, Focused KSM will merge the duplicate pages, and the result will be Region 2 be shared copy on write between Process A and Process B as shown in Figure 3.6.

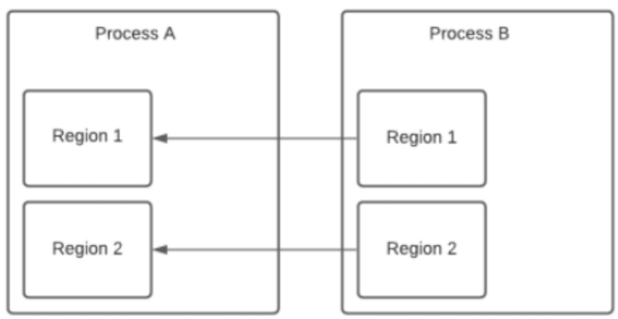

Figure 3.6: Processes A and B after Focused KSM has run. Region 1 was previously shared by Ultrafork and now Region 2 is also shared.

TODO: more details on this after it has been implemented.

3.2 Focused KSM (FKSM)

3.2.1 Initial Design

The initial design of FKSM uses 2 stages to deduplicate memory between 2 pids: traverse and merge. The traverse stage prepares 2 list data structures for the merge stage. Each structure contains the necessary metadata of all mergeable pages for a PID, and a Blake2b checksum of the page. The merge stage performs comparison lookups between these structures and merges when duplicate pages are identified. Further data structure optimizations are proposed and implemented in a second iteration design.

3.2.2 Traversing process memory

For the FKSM kernel module, we need to export the pagewalk API to `ioctl`s from within the kernel. This API allows FKSM to avoid re-implementing traversing of the page table hierarchy. The API utilizes callback functions for each page table level, but FKSM only needs to operate at the lowest per-page level. In the FKSM implementation, the callback used is `.pte_entry`. Within this callback, FKSM traverse performs two critical steps.

First, FKSM checks the page type and allows only anonymous pages to be operated on. This is for simplicity in the initial design, as the hashing algorithms chosen could perform poorly on huge pages. FKSM uses Blake2b due to it being a cryptographically secure algorithm that has higher bandwidth than SHA3. The bandwidth is still not high enough that we can confidently handle huge pages, so we elect to ignore them until a higher bandwidth algorithm is available. See Section 5.2 for discussion on improved algorithms.

Second, FKSM hashes the page and stores the checksum in a metadata structure. To get the checksum, FKSM uses `kmap_atomic` to map the address of the page to a local pointer. This pointer is then passed into Blake2b to compute the checksum. FKSM ends the critical section by unmapping the page. This metadata structure also contains the current page table entry, a pointer to the current `page` struct, and a pointer to the current `mm_struct` for the page. We store this metadata struct in the collection for the current PID being traversed for the merge operation later.

3.2.3 Merging the Pages

After FKSM is finished traversing the page table, it passes two lists of `page_metadata` structures to the function `combine`. This function iterates through the first list, gets the page from the current entry in the list and attempts to merge it with each other page in the second list. To merge the pages, FKSM checks that the page structs are not duplicates, then compares the Blake2b checksums of the two pages. If the checksums match, FKSM merges these two pages. In the minimum viable product, we perform the merge using the `replace_page` function that we have exposed from `ksm.c` in the kernel. Notably, this approach requires us to iterate over two lists, thus producing $O(n^2)$ performance, so we developed a major optimization on the minimum viable product, described below.

3.2.4 Data structure optimizations

To refine our minimum viable product, we organize our pages using a hash map of trees. This structure combines xxHash checksums with Blake2b checksums for each page to perform faster merge comparisons. These checksums are referred to as the Short Hash (sHash) and Long Hash (lHash) referring to xxHash and Blake2b respectively. This data structure starts with a layer 1 hashmap (L1HM) that is keyed off the first byte of the sHash. Each of these byte-keys will point to a layer 2 hashmap (L2HM) that contains the full 8-byte sHash. This L2HM is dynamically sized, and thus is organized as a doubly-linked list of "containers," each of which contains 32 buckets and pointers to the previous and next containers. The hash map implementation can create new containers as needed. The value for each sHash in the L2HM will be a pointer to a single Red-Black tree, which is keyed on the lHash.

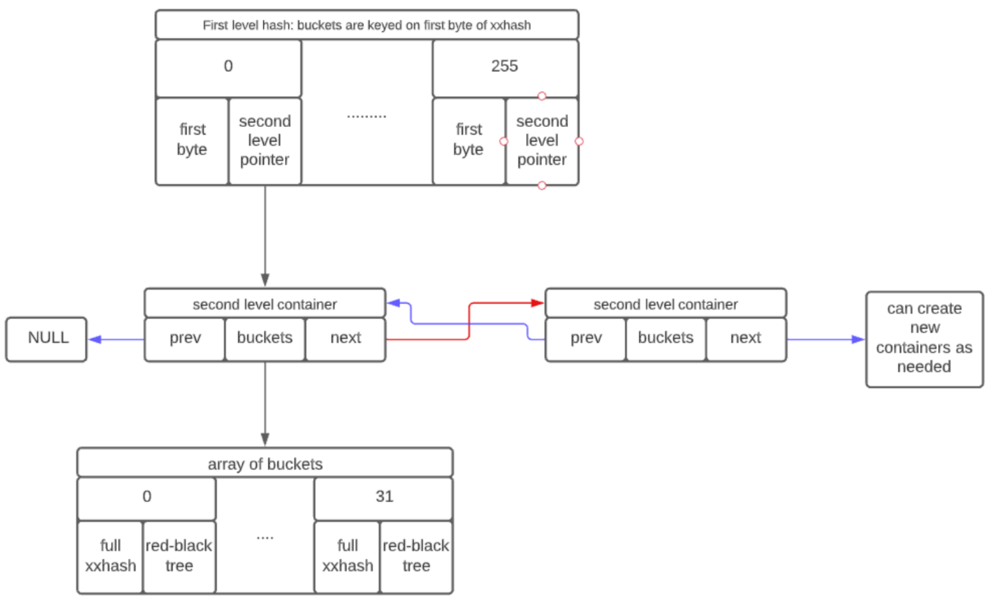

Figure 3.7: A diagram of the hash tree implementation, showing the levels of the tree starting from a single bucket.

When searching for a page, FKSM would use the first byte of its sHash in the L1HM to find which L2HM to look at next. Then, FKSM would use the rest of the sHash on the L2HM to find the appropriate red-black tree. We can finally use the lHash to search the red-black tree and identify any duplicate pages. The intent for this optimization is to improve the $O(n^2)$ performance of iterating over two lists that was done in the initial design. The indirection of having a L1H and L2H is because a single table of 8-byte xxHashes could result in buckets that are too large and have worse performance.

3.2.5 Restructuring for a Hash Tree

When switching to this new data structure, FKSM needs to perform both the traverse and combine steps in one scan function. In the previously-used page walk callback function, FKSM now performs its page type checks, then calls a helper function to generate both hashes of the current page. FKSM uses a get_or_create method to return any existing page metadata if the hashes are identical or insert the current page metadata into the hash tree if none is found. The condition for a successful insertion is a NULL pointer return. Otherwise, a non-null pointer references the metadata structure of an existing page we can merge the current page into. We then store the data needed to call `replace_page` at the tail of a linked list. After FKSM finishes walking the page tables, it then iterates over this list and calls `replace_page` with these stored values. The reason for performing this delayed page replacement is the page table entries are modified in `replace_page`. At one point in development, FKSM immediately called `replace_page` after finding an existing page. This led to a problem where the page tables suddenly changed in the middle of walking, causing a crash.

3.3 KSM / UKSM Test Instrumentation

To test our implementation of FKSM, we made a small kernel patch that added two new values to the sysfs entries for KSM and UKSM. The first value was `pages_merged`, which incremented each time the `replace_page` function was run in KSM and UKSM, as this was our main function we were using to merge pages in the minimum viable product. The second value was `pages_scanned`, which incremented on each iteration inside KSM's `do_full_scan` function, thus counting how many pages KSM had scanned.

3.4 Copy-on-Write (COW) Counter

Linux exposes a number of measurements for the amount of memory allocated to a process at a given time. The Virtual Set Size (VSZ) of a process is the amount of virtual memory allocated to the process (not necessarily loaded into memory) [3]. The tool `ps` can be used to view these measurements for a process with the command: `ps -o vsz <pid>`.

In order to evaluate Ultrafork, we need to be able to measure the total memory shared copy-on-write with a process. There is no way to measure this value on Linux systems. As part of this work, we created the COW Counter, a tool for measuring copy-on-write memory. The COW Counter uses an `ioctl` call provided by the SuS `ioctl` to determine the number of bytes shared with a given process. The userspace portion of the tool then takes this value, converts it to kilobytes and prints it in kilobytes, along with the virtual memory size (VSZ).

The COW Counter walks a process's `anon_vma_chain` in a similar manner as described in Section 3.1.4. First, we iterate through all virtual memory areas (`vm_area_struct`) for the process. At this point there is no way to determine if a given virtual memory area has been shared or is unique to this process. To determine this for each area, we walk the `anon_vma_chain`. The `anon_vma_chain` is populated with an element for each other process that shares this memory region. The COW Counter visits the first element on the chain, uses its `vma` pointer to get the size of the `vm_area_struct` and adds it to its running total of COW memory. Other elements on the chain do not need to be visited as they refer to the same memory region, but represent other processes sharing the same data. For example, if a virtual memory area in a grandparent process was shared with both the parent and child processes, the corresponding `anon_vma_chain` would have two elements. In addition to calculating the COW memory for a process, the COW Counter also computes the total virtual memory (VSZ) of a process. This is easily done while looping through the virtual memory areas of the process. Both values are returned to userspace in the context structure passed into the `ioctl`, which is shown in Listing 3.6.

Listing 3.6: Context structure for the COW Counter `ioctl`. The `pid` is the process ID of a process to measure COW memory of, `cow_bytes` and `vm_bytes` are used to return the measured values of COW memory and virtual memory respectively.

```
struct cow_ctx {
    pid_t pid;
    size_t cow_bytes;
    size_t vm_bytes;
};
```

After a userspace program calls the COW Counter, the context structure's `cow_bytes` and `vm_bytes` fields will be populated. The kernel returns these values in bytes, but the values are converted to kilobytes before being printed by the COW Counter. This is done for consistency with tools like `ps`, which report values in kilobytes. In general, the more closely related a process is to its parent, the larger fraction of its virtual memory will be shared copy on write. For example, a process forked from a parent that goes on to perform similar tasks in the same program will probably share a large amount of its memory, while a process spawned from a shell will share a much smaller amount. The COW Counter can work on any Linux process that is currently running, not just processes started through Ultrafork.

Chapter 4

Evaluation

4.1 Benchmarking the SuS System

Before we implemented Focused KSM and Ultrafork, we evaluated the memory usage of the Single Use Server (SuS) system and compared it with the memory usage of SuS when used with KSM and UKSM.

To evaluate the effectiveness of KSM and UKSM on the SuS system, we created a test infrastructure. The test infrastructure started an instance of the SuS system with a configurable number of containers. The test scripts then automatically visited the SuS webserver with a number of clients, causing web traffic. We collected the memory usage and page faults during the simulated network load and graphed them for analysis.

For these tests, we performed trials with 4, 8, 16, 32, 64 and 128 containers. Each trial had the same number of concurrent simulated "users" as the number of containers, as is intended by the SuS system. There was a random delay of 2-4 seconds between subsequent web requests to the same container to more realistically simulate a real user. URLs were randomly selected from a predetermined list of URLs valid in the SuS Wordpress installation. Each trial involved executed 500 web requests.

We tested three varying system conditions. The first case is a Ubuntu 20.04 5.14 kernel with no KSM or UKSM in play. We used the default Ubuntu kernel configuration file. The second configured is with KSM.

We recompiled the same kernel with KSM enabled. Then `LD_PRELOAD` was used to load a shared object that intercepted calls to `brk` and `mmap` to mark allocated pages with `MADV_MERGEABLE` as is required for KSM to consider those pages while scanning. All memory allocations performed by the SuS system went through this wrapper program.

The final configuration was UKSM. The same kernel was patched to provide UKSM support, and was recompiled. Marking pages as `MADV_MERGEABLE` is not required for UKSM to function, so the wrapper program was removed.

4.1.1 A Note on Major Page Faults

When running tests, we rebooted the test machine between configurations. Due to this, the major page faults were abnormally high for the first set of container tests as the system gathered required pages. The result of our analysis is that major page faults produced no significant observable data. See below for an example of what the graph looked like for the KSM trials. The first tests run used 4 containers and this test always had the most significant share of the page faults. UKSM shared a similar looking graph.

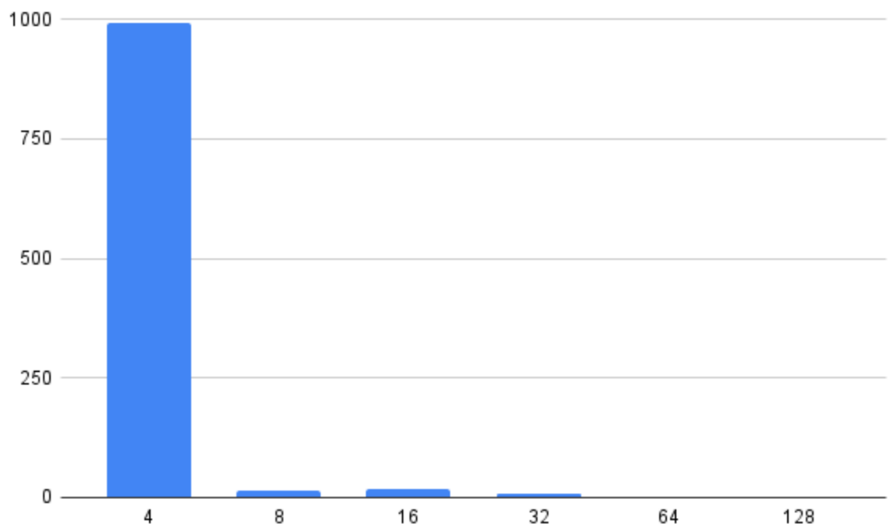

Figure 4.1: Major page faults with a varying number of SuS containers when KSM is active.

4.1.2 Control Test Results

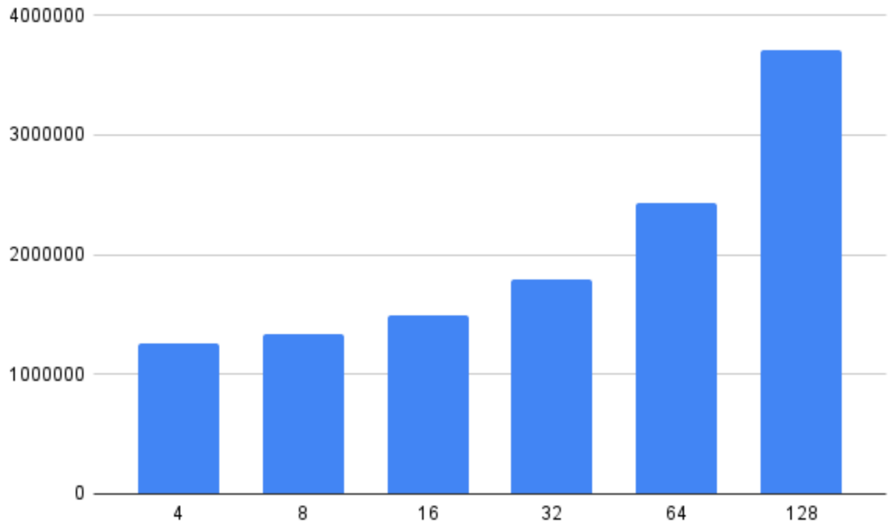

Figure 4.2: Memory Usage (Kilobytes) of the system with a varying number of SuS containers.

Control Memory Usage

In figure 4.2 the memory usage increased with the number of containers. The slope of the increase in memory usage appears to increase because the number of containers was doubled in each trial. These levels of memory usage were the highest, as seen in our later tests.

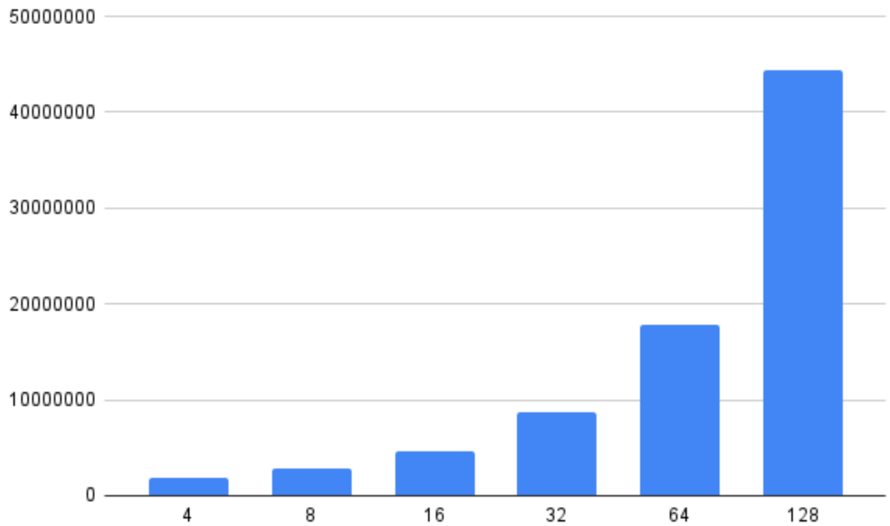

Figure 4.3: Minor page faults of the system with a varying number of SuS containers.

Control Minor Page Faults

In figure 4.3 we can see that the numbers of minor page faults increased with the number of containers. As mentioned above, the slope appears curved only because the number of containers doubled each trial. These page fault levels were the highest, as seen in our later tests.

4.1.3 KSM Test Results

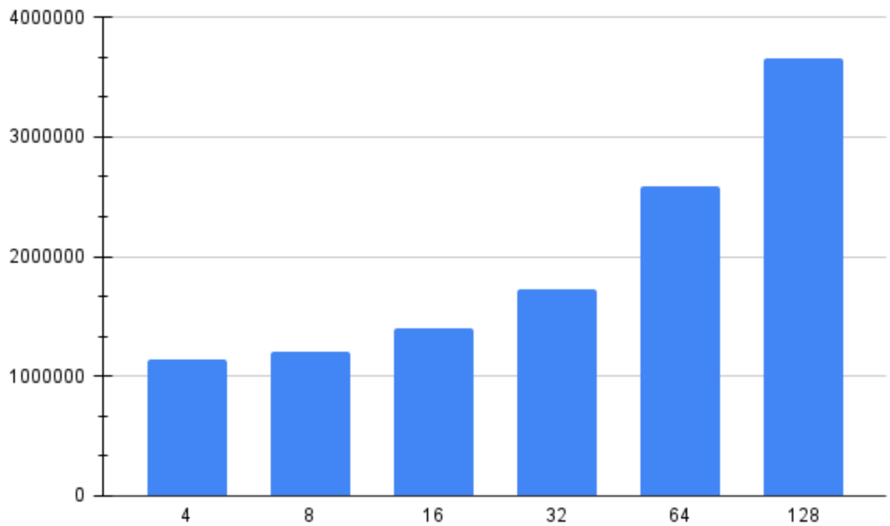

Figure 4.4: Memory usage with a varying number of SuS containers when KSM is active.

KSM Memory Usage

The KSM tests in figure 4.4 showed the same trends in memory usage, as expected, but with some improvement. The improvement is small, but it shows that KSM does provide some benefit to memory usage.

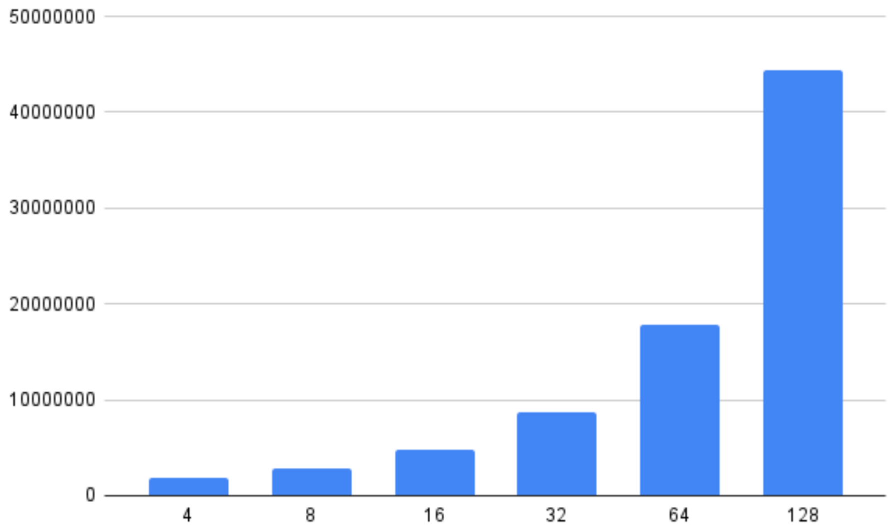

Figure 4.5: Minor page faults with a varying number of SuS containers when KSM is active.

KSM Minor Page Faults

We can see in figure 4.5 that KSM provided very little benefit to minor page faults. The numbers of minor page faults under KSM were almost identical to control and followed the same trend.

4.1.4 UKSM Test Results

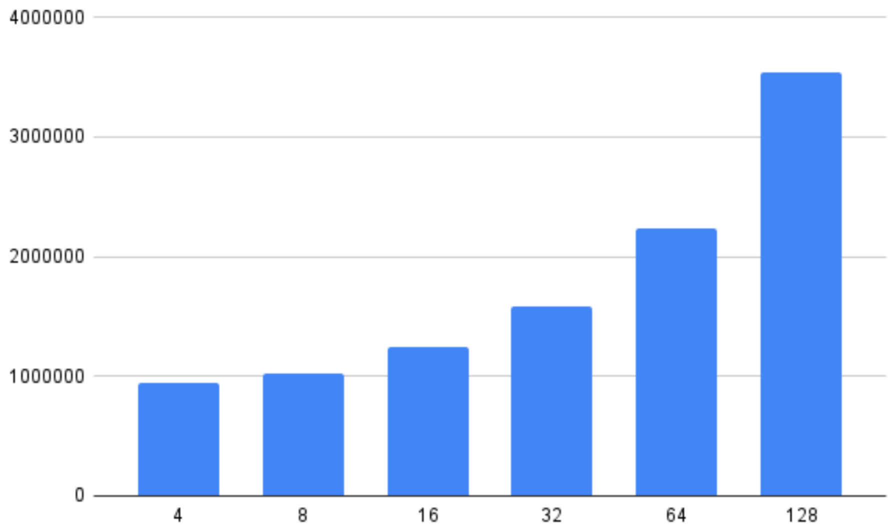

Figure 4.6: Memory usage with a varying number of SuS containers when UKSM is active.

UKSM Memory Usage

Figure 4.6 shows that UKSM provided a small but more noticeable benefit to memory usage. The memory usage, as expected, followed the same trend as the control, but with a larger improvement. This shows that, at least in this test environment, UKSM performed better than KSM.

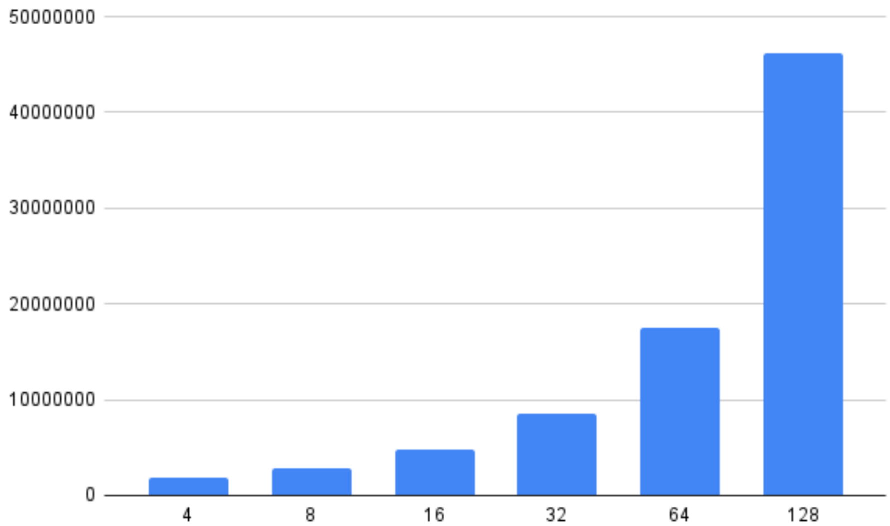

Figure 4.7: Minor page faults with a varying number of SuS containers when UKSM is active.

UKSM Minor Page Faults

In figure 4.7 we see that UKSM provided a slight reduction in minor page faults. This and the memory improvements combine to show that UKSM is an effective improvement over KSM.

4.1.5 Commentary

As expected, the memory usage and page faults for the control set, KSM and UKSM increased linearly as the number of containers increased, as did the minor page faults. Additionally, UKSM showed smaller overall levels of memory usage and minor page faults. The KSM and UKSM tests showed large spikes of major page faults during their first trials, however we have determined that these were caused by the system having just restarted, so there were a string of page faults as the system gathered the pages needed.

4.2 Focused KSM

Our testing and evaluation of FKSM is structured as a parent process which forks a child process. Each process will allocate some number of page-aligned 4KB regions, called "test pages". This number of test pages allocated for both parent and child is the Total-Test-Pages (TTP). The pages per process is called the Pages-Per-Process (PPP), and is equivalent to half the TTP. To facilitate the `ioctl` call of FKSM, the processes also share memory using `shmget()` and `shmat()`. They store their PIDs and pointers to the test pages in this shared memory region so the parent can call the `ioctl` with both PIDS. We also use the test page pointers to generate a long hash in userspace as a means of validating the kernel long hash is correct. After allocating the test pages, both processes sleep for 2 seconds to wait for FKSM. They then release their pages and exit.

In FKSM, a "scan" is characterized as the combined short/long hash operation and `get_or_create()` of an anonymous page in the page table walk callback. After all pages are scanned, we perform the `replace_page` loop (see Section 3.2.5). The scan performance of FKSM is almost entirely based on how quickly the hashing and lookup operations can occur for a given page. The `replace_page` loop performance is based on how many duplicates were identified in the scan and the length of time `replace_page` takes to execute. The linked list used is as simple as possible to contribute minimally to the overall loop time.

All tests were conducted at a TTP of 4, 100, 200, 2000, 5000. PPP is therefore 2, 50, 100, 1000, 2500. Total test data merged is also therefore 16KB, 400KB, 800KB, 8000KB, 20,000KB. Results are the average of 10 trials

Table 4.1: Total number of scans and total time for FKSM overall. TTP is listed as both PIDs are operated on

TTP	Data (KB)	Scans	Merges	Total Time (ms)
4	16	58	12	3.36
100	400	250	156	14.22
200	800	451	306	27.01
2,000	8000	4,050	3,006	209.15
5,000	20,000	10,050	7,506	542.35

Analysis of these results shows that for a given TTP, there are approximately 2*TTP scans conducted. This runs counter to the expected behavior that pages should only be scanned once. FKSM walks the page tables of the parent first and the child second, theoretically scanning $\approx$ 2*PPP times which should be $\approx$ TTP. This indicates a double scan is likely occurring. We theorize that there is room to improve performance by addressing this bug and scanning pages optimally. The appearance of $\approx$ 50 extra scans per run indicates code and other program data pages are being scanned in addition to the test data. Following the previous theory of a double scan bug, it follows that there should be ≈ 25 code pages across both processes or approximately 12-13 code pages per process.

The total running time of FKSM is over half a second for 5000 total test pages. For the sake of calculating the bandwidth of the system, we choose the largest number of pages to reduce the impact of constant factors in setup and exit of FKSM. Including 25 extra code pages as described previously, all of size 4096 bytes, FKSM scanned ≈ 19.63 MiB of real data in 0.54235 seconds. This is roughly equivalent to a bandwidth of 36.2 MiB/s. This estimated bandwidth is initially underwhelming, but with described improvements in Section 5.2 and the resolution of the double scan bug, results should improve by a significant factor. Fixing the double scan alone would be a two times improvement.

Table 4.2: Average time for combined sHash/lHash operations and lookup time of a scan

TTP	Hash Time (us)	Lookup Time (us)
4	23.7	19.4
100	24.7	15.6
200	26.1	16.1
2,000	23.6	11.5
5,000	24.3	12.9

In testing, the average lookup time decreased as the number of test pages increased. More “pages under management” should improve the average hash tree performance, as the results show. Additionally as expected, the average execution time for the sHash/lHash operation deviated very little based on the TTP. This hashing operation should be relatively constant for the given page size of 4096 bytes. See Section 5.2 for proposed improvements to hash speed.

4.2.1 Bandwidth comparison with USKM

To compare FKSM in its current state to UKSM, we calculate the total bandwidth as the number of pages scanned over the test duration. This is the same concept as FKSM’s bandwidth analysis, except the test data comes from real container tests described in 4.1 instead of simple processes. To arrive at bandwidth values, we determine the total number of pages scanned over the duration of the test using the UKSM entries in the `sysfs`. We then divide the total scanned pages over the duration of the test to arrive at a rate of pages/second. Our final step is to multiply this approximate page rate by 4096 to get a value in some number of bytes per second.

Table 4.3: Bandwidth calculation for container tests of UKSM with container count and test duration

Number of Containers	Bandwidth (MiB/s)	Test Duration (Minutes)
4	3.8	4.1
8	2.2	4.2
16	0.99	4.4
32	0.16	4.7
64	0.49	5.9
128	0.19	12.1

From a raw numbers perspective, the most charitable comparison between FKSM and UKSM is a 9.5x higher bandwidth in favor of FKSM, but there are many caveats to this comparison. These tests were not done on the same system, or on the same sample processes. The UKSM test concluded when all processes were finished, not when UKSM was done scanning the processes. Additionally, UKSM scans the whole system, not just 2 processes. We expect the numbers to be inflated in favor of UKSM with more processes to merge, but tuning parameters reduce performance to play nicely with the rest of the system. A direct comparison is necessary to know which KSM derivative has the higher bandwidth. Does a greedy approach to memory deduplication beat a generalized and tuned system-wide approach? We can not say for certain one way or the other. The only inference to be made is that FKSM is more “bursty” and UKSM is more “measured” in their approaches to same-page merging.

4.3 Ultrafork

There are two logical ways to evaluate the effectiveness of Ultrafork. The first is to demonstrate how copy-on-write can result in memory savings between processes. The second is to measure the performance cost of Ultrafork and compare it to `fork`. The following sections perform these measurements.

4.3.1 Copy-on-Write (COW) Memory

The main objective of Ultrafork is to reduce the memory overhead of groups of related processes, which it does by using copy-on-write to share memory. A logical measurement of the effectiveness of Ultrafork is to measure how much of the cloned process’s virtual memory is copy on write from its parent. To do this, we created the COW counter, as described in Section 3.4.

We wrote a simple C program to establish a baseline of copy-on-write memory between two closely related processes. The program spawns a child process, and then calls Ultrafork, which clones both the main process of the program and the spawned child process. Then the COW counter is run on the two cloned processes. Table 4.4 shows the results of this test.

Table 4.4: A baseline measurement of memory COW shared between processes used in the test.

Process	COW Memory (kB)	Virtual Memory (kB)
Original Parent Process	236	2,648
Original Child Process	368	2,648
Cloned Parent Process	368	2,648
Cloned Child Process	368	2,648

The table shows a parent and child and their clones after an Ultrafork is run on the original parent process. Each process calls `sleep` and then exits. Minimal setup is done before the Ultrafork call. Each process has about 2 MB of virtual memory and 368 kB of COW shared memory, which is most runtime data for the program and its dependencies (only glibc in this case).

To illustrate the effectiveness of Ultrafork, we modified the test program to include an 8 GB `malloc` call in the original parent process before the Ultrafork operation. As before, the Ultrafork operation was executed and each process slept before exiting. During the `sleep` call, the COW counter was run on each process and results are shown in Table 4.5. Before forking the original child process, the original parent process has 236 kB COW memory shared with its parent, which is a `bash` shell. This COW memory is likely made up of code and data from glibc. The original child process shares a bit more memory with its parent, which makes sense since the processes share very similar code and data. Things get interesting when we perform the large memory allocation and call Ultrafork. After Ultrafork, the cloned parent process shares over 8 GB of data with the original parent process, and its virtual memory increases accordingly. The cloned child process shares the 368 kB with its parent, the original child process. This experiment demonstrates that Ultrafork is effective at sharing data with child processes.

4.3.2 Time Cost

Ultrafork has more overhead when compared to `fork()` because of the setup and re-parenting phases. Additionally, Ultrafork must clone a number of tasks instead of `fork()` cloning only a single task. We compare the time cost of Ultrafork to the time cost of the same number of `fork()` operations Ultrafork performs internally to make a fair comparison.

Table 4.5: Memory in kB shared with a parent process after an 8 GB allocation in the original parent process. The Ultrafork operation happens after the allocation, so the memory is shared between the original parent process and the cloned parent process. The child process does not contain this allocation.

Process	COW Memory (kB)	Virtual Memory (kB)
Original Parent Process	236	2,516
Original Child Process	368	2,648
Cloned Parent Process	8,388,980	8,391,260
Cloned Child Process	368	2,648

To determine how costly Ultrafork is compared to `fork()`, we devised a mechanism to benchmark `fork()`. To measure the kernelspace time spent on a `fork()` call, we implemented a `kretprobe`. The probe was attached to the `kernel_clone` function, which performs the majority of the work for the `fork()` operation. The only work not included in this call is the initialization of the clone arguments structure, which is a negligible contribution to the total time. The `kretprobe` uses the same clock we used to benchmark Ultrafork, the scheduler clock, for consistency. Measurement starts when the `kernel_clone` function starts executing, and stops when the `kernel_clone` function returns. Note that this time excludes the time taken to cross the kernelspace and userspace boundary (which is significant). This time is excluded to remain consistent with the benchmarks of Ultrafork.

To use the `kretprobe`, we wrote a simple userspace program to fork a number of processes according to a command line argument. The source for this program is shown in Listing 4.1. In the test cases, we execute the program repeatedly from a script, passing an increasing number of processes. No blocking or waiting is necessary in this program since the only measurement is the time taken for each fork. After the fork, the forked process is free to exit. We benchmarked forking 1 to 100 processes, with 10 trials at each number of processes. The script was also responsible for recording the time spent executing the `fork` call, which the kretprobe printed to the kernel log.

Listing 4.1: Simple program to benchmark `fork()`. Accepts a command line argument to determine how many processes to fork.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
    int count = atoi(argv[1]);
```

```
    if (count < 1) {
        exit(1);
    }
    pid_t p = fork();
    if (p != 0) {
        for (int i = 1; i < count; i++) {
            pid_t x = fork();
            if (x == 0) {
                break;
            }
        }
    }
    return 0;
}
```

Similarly, we implemented a test program to perform Ultrafork on a number of processes specified by a command line argument. As before, a script repeatedly called the Ultrafork userspace test program with an increasing number of processes, from 1 to 100. A total of 10 trials were run at each number of processes. We computed the average time for each number of processes and created a cumulative distribution function (CDF) plot to illustrate the differences between Ultrafork and `fork`, as shown in Figure 4.8. The CDF can be used to compare the differences between the Ultrafork and `fork` benchmarks. If Ultrafork had identical performance to `fork`, we would see two lines exactly overlaid on the plot. If the Ultrafork line appears above the fork line, that indicates that Ultrafork is slower than `fork` at that value. The plot shows that when dealing with a small number of processes (10 or less), Ultrafork and `fork` have similar time cost. The time difference between Ultrafork and `fork` is less than 1% when dealing with 5 or fewer processes and less than 2% when 10 processes are involved. The greatest difference between Ultrafork and `fork` is when dealing with 45 processes. In the trials with 45 processes, Ultrafork was 13% slower than `fork`. At 46 processes, the difference between Ultrafork and `fork` starts to decrease. When dealing with 95-100 processes, the overhead of Ultrafork drops below 3%.

Ultrafork performs well compared to fork for a relatively small number of processes (10 or fewer). Ultrafork starts to show significant overhead, greater than 10% beginning with 29 processes and then drops back below 10% at 64 processes. This is most likely due to the cost incurred by crossing the userspace-kernelspace barrier many times for `fork`, and only once for Ultrafork. It makes sense that as the number of processes increase, the cost of a large number of `fork`s starts to become visible.

It is important to note that there is an inherent noise level in these benchmarks, since the `fork` and Ultrafork operations are both interruptable. At any time, the kernel could schedule another task in place of the benchmarked tasks, adding error to the measurement. The average of 10 trials at each number of processes should help mitigate the noise in these measurements.

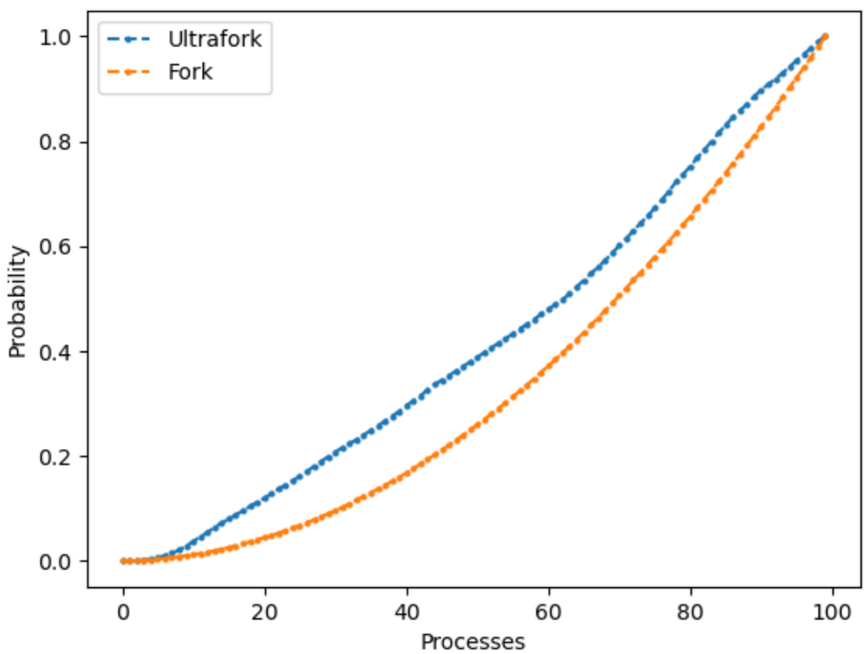

Figure 4.8: Cumulative Distribution Function (CDF) graph comparing Ultrafork and Fork for a variable number of processes. Each data point is the average of 10 trials.

Chapter 5

Conclusion and Future Work

While we implemented Ultrafork and Focused KSM, we identified a number of areas for potential improvement and extensions, which we discuss in the following sections.

5.1 Ultrafork

The main area of future work for Ultrafork is the support of threads. Currently, Ultrafork cannot clone threads. We attempted an implementation of thread cloning, but did not complete it. Threads pose a more difficult problem than processes, since processes are designed to be cloned on Linux systems. There is no mechanism to clone threads in a similar way to Ultrafork, so the implementation is much more difficult. Thread support would be quite desirable, as it would allow most programs to be transparently supported by Ultrafork. Currently, cloned threads crash immediately when they wake up in userspace.

Another area of improvement for Ultrafork is the addition of support for a subset of `clone` flags. As explained in Section 3.1.8, Ultrafork sets several of these flags internally to control the kernel's clone operation. Ultrafork could be extended to allow users to specify any of the flags not used internally by Ultrafork. The user specified flags would be supplied to the `clone` call for each process Ultrafork cloned. The caller would be responsible for ensuring the flags cause the desired effect when applied to multiple processes.

Before Ultrafork clones processes, it must prevent the target processes from executing to ensure consistency. Failure to properly lock one or more of the processes could result in a crash or in data corruption. The current technique used to prevent data races during Ultrafork is to send `SIGSTOP` to each process involved in the Ultrafork operation. After the processes are cloned, they are resumed with `SIGCONT`. This approach works, but has one major drawback: if a process installs a signal handler to ignore or temporarily ignore `SIGSTOP`, Ultrafork will not properly suspend the running processes. Future work could forcibly prevent the processes from being scheduled for the duration of the Ultrafork operation.

Work could be done to optimize the Ultrafork, particularly when starting and stopping processes. This could be optimized be informing the scheduler that the processes Ultrafork needs to operate on should not be scheduled until after Ultrafork completes. This would have the same effect as the current method but would not have the overhead incurred by signal delivery.

5.1.1 Thread Support

Linux's fork implementation only copies the caller's thread of execution. If a multi-threaded program is forked, only the thread that initiates the fork is copied into the new process. The caller is responsible for ensuring consistency with locking constructs, usually with the `pthread_atfork` call. Threads must be supported in order for Ultrafork to function properly on any given program. Contrary to `fork()`, Ultrafork should make clones of each thread for each process it clones, resulting in a complete copy of the given program. Since `fork()` does not support multiple threads, this functionality must be implemented from scratch.

The `task_struct` contains a number of process ID fields for various purposes, all of which need to be correctly adjusted to handle threads. The Thread Group ID (tgid) allows threads that are part of the same process to have a single shared identifier. All threads in a process must have the same value, which is equal to the PID of the process that originally started those threads. The `task_struct pid` element is not the Process ID (PID) as it is seen from userspace, but actually a thread ID (tid). This allows tasks to be uniquely identified with the `pid` field. A simple method to test if a task is a thread or a process is to test `task->pid == task->tgid`, which will be true if the task is a process.

A difference between threads and processes is the usage of the `parent` and `real_parent` pointers in `task_struct`. In a process, both `parent` and `real_parent` point to the process that should receive `SIGCHLD`. This is typically the process that spawned this process, or the process that was it was re-parented to in the event that the original parent process died. In a POSIX thread, the `real_parent` instead points to the task of the parent thread [6, p. 16].

Re-parenting is a complex operation for threads. For processes, the copy-on-write tendency of most memory makes the operation fairly simple. Threads share virtual address space with their parent process, which complicates matters considerably. After the clone phase of Ultrafork, a cloned thread shares address space with the process that spawned the original thread. In the re-parenting phase, the cloned thread needs to be made to share address space with the cloned version of the process that spawned it. This is done using a modified version of the `copy_mm` function used in `fork()`. Additionally, threads share signalling information with their parent process, which is stored in both the `task->signal` and `task->sighand` structures. Ultrafork assigns the cloned thread's `signal` and `sighand` pointers to the values owned by the new parent process. Reference counts are incremented in the new parent process and decremented in the old parent process to ensure consistency after the operation completes. Since the thread is being moved to another parent process, there is the potential for credentials to be different between the two processes. In Linux, credentials are the kernelspace mechanism for handling permissions, capabilities, keys and other security related features [13]. Ultrafork needs to update the thread's credentials from those of its old process to those of its new process. This is done through a modified version of the `copy_creds` call.

We attempted to implement support for threads in Ultrafork, but were unsuccessful. The implementation was able to start cloned threads, but a segmentation fault occurred in userspace when the cloned threads awoke. This is likely due to the complex nature of thread local storage (TLS). TLS is implemented on Linux with a dedicated section of heap space for each thread, accessed through the FS register (on x86_64) [9]. The userspace library pthread is responsible for managing these regions of memory. When a new pthread is started, the userspace library sets aside memory for the TLS region, and then starts the thread using the `clone` system call. Ultrafork needs to somehow set aside new memory regions for TLS of cloned threads, possibly using a modified version of pthread. In addition to TLS, pthread also keeps metadata about running threads in the process's virtual memory. This metadata includes cached copies of a thread's TID for faster access than the `gettid` system call.

Due to the unimplemented thread support, if Ultrafork is called on a process with threads, or a process with descendants that have threads, an error will be returned, and a message will be logged in the kernel log at the error level indicating the process ID and thread ID of the offending thread. This message can be used for debugging to determine what processes inside a container are starting threads. If thread support is added in the future, this functionality should be removed.

5.1.2 IPC problems

Ultrafork presents an interesting problem with common IPC constructs, such as System V semaphores and shared memory regions. If a semaphore exists between two processes that are Ultrafork, there will then be four users of the semaphore, and the potential for race conditions or deadlocks exist.

Shared memory regions do not inherently pose a problem on their own, but race conditions could occur if the region is written to by multiple processes without proper locking. To address this issue, Ultrafork will detect the condition where IPC constructs are in use, and fail if any constructs are currently in a 'locked' state. Constructs that are not currently in a 'locked' state can be safely replicated without causing race conditions. This feature would be easy to implement, since IPC constructs are tracked in the `task_struct`.

5.2 Focused KSM

Focused KSM is currently implemented as a call through an `ioctl`. The design of this call is providing 2 PIDs, but FKSM can theoretically support a list of PIDs and count of list elements instead. This allows the code to be modular and easier to call, only requiring a patch to expose the `replace_page()` symbol in the Linux KSM source.

Further work could also involve creating a kernel thread that maintains the hash tree structure as described in Section 3.2.4. This structure will perform optimally over multiple runs where duplicate checksums are identified and pages replaced without insertion. Part of this future work into a kernel thread is a clean up process for the structure when processes exit. Deleting references to pages that no longer exist keeps our hash tree up to date and as small as possible. Work would also be directed at the scheduling behavior for this kernel thread implementation. Problems such as when to scan a process after Ultrafork, if a process is in an idle versus active state, and what the optimal interval for repeat scans is are all questions that can be brought up as tuning parameters.

Optimizations of the hash tree implementation are possible with more efficient traversal of the red-black trees in `get_or_create()`. As written, the code attempts an insertion onto the tree, finds a collision, then performs a separate `rb_search` to find the data again. A method to insert or return a colliding node would reduce tree traversals.

Lastly, we can replace Blake2b with Blake3 for the Long Hash (lHash) in our hash tree structure. The Blake3 algorithm is 5x faster than its predecessor, but no kernel implementation yet exists at the time of writing this paper [19]. If the developers of Blake3 provide an implementation or a future group would like to attempt development of this implementation, it would surely speed up the lHash process in the page traversal phase of FKSM.

5.3 Conclusion

In this work, we presented Ultrafork and Focused KSM, two strategies for optimizing memory in large numbers of similar Docker containers used in the SuS system. We implemented Ultrafork, a technique for forking groups of processes, allowing memory to be shared copy-on-write from the beginning of a container's lifetime. Focused KSM offers a way to merge duplicate pages that arise during the course of a process's execution. Together, they implement a system that effectively shares memory without significant CPU costs. We say that Ultrafork has a 13% overhead in the worst case, with a less than 2% overhead when dealing with fewer than 10 processes. We found that Ultrafork can facilitate a large amount of memory sharing between processes depending on how similar they are.

Focused KSM offers a simpler method of calling memory deduplication on pairs of processes, with mixed performance statistics in its current form. As the number of pages increases for real-world applications and with all the proposed optimizations, FKSM has the potential to be a lightweight maintainer of the Ultrafork memory savings. FKSM and Ultrafork work together by reducing the scanning overhead and reducing calls to deduplicate memory.

Although originally intended for containers, Ultrafork and Focused KSM could be used in a variety of other contexts, including with processes that are not containerized. A potential use case for this would be to efficiently start groups of worker processes for a large application. Instead of forking worker processes in a loop, Ultrafork could be utilized to reduce the overhead of multiple kernelspace-userspace boundary crosses. Focused KSM could then be used on the worker processes to limit their memory impact.

Finally, when evaluating the performance of UltraFork, we implemented the COW Counter, a tool for measuring copy-on-write memory of a process. This tool could be used to evaluate the effectiveness of similar systems in the future, or to identify new potential applications for Ultrafork.

Bibliography

[1] Marcelo Abranches et al. "Shimmy. Shared Memory Channels for High Performance Inter-Container Communication". In: (2019). URL: https://www.usenix.org/system/files/hotedge19-paper-abranches.pdf.

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. "Increasing Memory Density by using KSM". In: *Linux Symposium* (2009). URL: https://www.kernel.org/doc/ols/2009/ols2009-pages-19-28.pdf.

[3] procps-ng authors. *ps utility man page*. Feb. 19, 2022. URL: https://manpages.ubuntu.com/manpages/jammy/en/man1/ps.1posix.html.

[4] The gVisor Authors. *gVisor*. Nov. 15, 2021. URL: https://gvisor.dev/.

[5] Paul Barham et al. "Xen and the Art of Virtualization". In: *ACM SIGOPS Operating Systems Review* (2003). URL: https://doi.org/10.1145/1165389.945462.

[6] Raghu Bharadwaj. *Mastering Linux Kernel Development*. Packt Publishing, 2017. ISBN: 978-1-78588-305-7.

[7] Thanh Bui. "Analysis of Docker Security". In: *Aalto University Seminar on Network Security* (2014). URL: https://arxiv.org/pdf/1501.02967.pdf.

[8] Xian Chen et al. "SEMMA. Secure Efficient Memory Management Approach in Virtual Environment". In: *International Conference on Advanced Cloud and Big Data* (2013). URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6824585.

[9] The Linux Kernel Development Community. *Using FS and GS segments in Userspace Applications*. Feb. 27, 2022. URL: https://docs.kernel.org/x86/x86_64/fsgs.html?highlight=thread%5C%20local.

[10] Grzegorz Czajkowski, Laurent Daynes, and Nathanial Nystrom. "Code Sharing Among Virtual Machines". In: (2002). URL: http://www.cs.cornell.edu/nystrom/papers/cdn02-ecoop.pdf.

[11] Dong Du et al. "Catalyzer. Sub-millisecond Startup for Serverless Computer with Initialization-less Booting". In: *ASPLOS* (2020). URL: https://dl-acm-org.ezpv7-web-p-u01.wpi.edu/doi/pdf/10.1145/3373376.3378512.

[12] Diwaker Gupta et al. "Difference Engine. Harnessing Memory Redundancy in Virtual Machines". In: (2010). URL: https://dl.acm.org/doi/pdf/10.1145/1831407.1831429.

[13] David Howells. *Credentials in Linux*. Feb. 22, 2022. URL: https://www.kernel.org/doc/html/v5.16/security/credentials.html.

[14] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. "Determining the use of Interdomain Shareable Pages using Kernel Introspection". In: (2007). URL: http://mejlholm.org/uni/pdfs/dat7_introspection.pdf.

[15] H. Andres Lagar-Cavilla et al. "SnowFlock. Rapid Virtual Machine Cloning for Cloud Computing". In: (2009). URL: https://dl.acm.org/doi/pdf/10.1145/1519065.1519067.

[16] Julian P. Lanson. "Single-Use Servers. A Generalized Design for Eliminating the Confused Depty Problem in Networked Services". Thesis, Worcester Polytechnic Institute, 2020.

[17] Robert Love. *Linux Kernel Development*. Addison Wesley, 2010. ISBN: 978-0-672-32946-3.

[18] Grzegorz Milos et al. "Satori. Enlightened Page Sharing". In: (2009). URL: https://www.usenix.org/legacy/event/usenix09/tech/full_papers/milos/milos_html/.

[19] Jack O'Connor et al. *BLAKE3. one function, fast everywhere*. Feb. 24, 2022. URL: https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf.

[20] Edward Oakes et al. "SOCK. Rapid Task Provisioning with Serverless-Optimized Containers". In: *USENIX Annual Technical Conference* (2018).

[21] Mehrnoosh Raoufi et al. "PageCmp. Bandwidth Efficient Page Deduplication through In-memory Page Comparison". In: *IEEE Computer Society Annual Symposium on VLSI* (2019). URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8839416.

[22] Ziming Shen et al. "X-Containers. Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers". In: *ASPLOS* (2019). URL: https://dl.acm.org/doi/pdf/10.1145/3297858.3304016.

[23] Jonathan Smith and Gerold Maguire. "Effects of copy-on-write memory management on the response time of Unix fork operations". In: *Columbia University, Computer Science Department* (1988). URL: https://www.cis.upenn.edu/~jms/cw-fork.pdf.

[24] Yuqiong Sun et al. "Security Namespace. Making Linux Security Frameworks Available to Containers". In: *Usenix Security Symposium* 27 (2018). URL: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-sun.pdf.

[25] Kuniyasu Suzaki et al. "Memory Deduplication as a Threat to the Guest OS". In: *EUROSEC '11: Proceedings of the Fourth European Workshop on System Security* (2011). URL: https://dl.acm.org/doi/10.1145/1972551.1972552.

[26] Dmitrii Ustiugov et al. "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots". In: (2021).

[27] Michael Vrable et al. "Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm". In: *Proceedings of the twentieth ACM symposium on Operating systems principles* (2006). URL: https://dl.acm.org/doi/abs/10.1145/1095810.1095825.

[28] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. "Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment". In: (). URL: https://dl.acm.org/doi/pdf/10.1145/3302424.3303978.

[29] Wei Wang et al. "Reg: An Ultra-Lightweight Container that Maximizes Memory Sharing and Minimizes the Runtime Environment". In: *2019 IEEE International Conference on Web Services* (2019). URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8818398&tag=1.

[30] Zhigang Wang et al. "Dynamic Memory Balancing for Virtualization". In: *ACM Trans. Archit. Code Optim.* (2016). URL: https://dl.acm.org/doi/10.1145/2851501.

[31] Nai Xia et al. "UKSM. Swift Memory Deduplication via Hierarchical and Adaptive Memory Region Distilling". In: *USENIX Conference on File and Storage Technologies* 16 (2018), pp. 1–16. URL: https://www.usenix.org/system/files/conference/fast18/fast18-xia.pdf.

[32] Lingjing You et al. "Leveraging Array Mapped Tries in KSM fro Lightweight Memory Deduplication". In: *IEEE Conference on Networking, Architecture and Storage (NAS)* (2019). URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8834730.