On-Stack Replacement Across User-Kernel Boundaries

by

Katherine Mohr

B.S. Electrical Engineering and Computer Science, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024

Authored by:	Katherine Mohr Department of Electrical Engineering and Computer Science August 16, 2024
Certified by:	Saman Amarasinghe Professor of Electrical Engineering and Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

On-Stack Replacement Across User-Kernel Boundaries

by

Katherine Mohr

Submitted to the Department of Electrical Engineering and Computer Science
on August 16, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

In large, distributed computations with small amounts of work done at each node, networking latencies quickly add up, especially in comparison to the time taken to execute small tasks. As such, lowering network latencies is crucial to getting good performance. Previous research has shown that often the largest contributors to network latencies are data copies between kernel and application buffers. Conventional wisdom argues that to solve this problem, one should move the networking stack out of the kernel and into the user space or networking hardware. Instead, we build upon an alternative approach, known as LakePlacid. LakePlacid mitigates the kernel-user boundary overhead issue by moving the most important application logic out of the user space and into the kernel. This thesis proposes and implements a key improvement to LakePlacid. Because only part of the application logic is migrated to the kernel, some packets necessarily must be resolved in the standard user space application. The system discussed in this thesis allows packets which cannot be handled in the kernel to seamlessly continue in user space via on-stack replacement, thus preventing side effects from being executed erroneously. This system for on-stack replacement is very general, allowing execution to switch between code versions at any conditional, and it is novel in its ability to switch stacks across the user-kernel boundary. With this change, LakePlacid is able to better maintain the semantics of user applications, making it more feasible in practice.

Thesis supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I'd like to give my greatest thanks to my advisor, Saman Amarasinghe, and my mentor, Ajay Brahmakshatriya. Thank you to Saman for always having my back, helping me learn how to define and solve new problems, and encouraging me to go further in my academics. Thank you to Ajay sitting with me through many debugging sessions, for being understanding of my mistakes and frustrations and helping me to resolve them, for reassuring me when I felt discouraged, for teaching me so much about computer systems both conceptually and in practice, and for introducing me to this problem domain in the first place. The lessons I've learned from the two of you have greatly shaped my approach to research, and I'm very grateful to have been able to work with the both of you for the past couple of years.

While not directly involved in this project, I would also like to give a huge thanks several other faculty and staff at MIT who have contributed immensely to my growth as a student and researcher. Thank you to my first research advisor, Jonathan Ragan-Kelley, for helping boost my confidence in his lab and giving me the courage and motivation to continue with compilers research. Thank you to Michael Carbin for teaching me about compiliers initially and encouraging me to help teach multiple programming languages classes with you. Thanks as well for letting me intrude on PSG; I might not be a member of PSG, but thanks for considering me to be part of PSG+. I know the next person has to see this document, and I've heard the acknowledgements section is her favorite part, so thank you so much to Katrina LaCurts for giving me advice throughout my undergrad and making me feel like someone in the course 6 department was always looking out for me.

Thank you to so much to all the graduate students I've worked with (formally and informally) over the years, especially to everyone in the COMMIT, VCLS, and PSG research groups, and to the Tea Time and GSB regulars. You all have been so welcoming and made me excited to head into Stata everyday.

Thank you to my friends for all of the coworking sessions that devolved into chatter, for the coffee runs and sweet treat breaks, for the many excursions into Boston, and most of all, for the unwavering support you've provided me with. You all have made the past five years incredible, and I wouldn't have traded it for a thing. I am especially grateful to AJ Root. Thank you for being an extraordinary mentor early on in my undergrad career, even taking time out of your breaks to teach me about domain specific languages and performance engineering, and thank you for continuing to be my biggest hypeman today.

Finally, I'd like to thank my parents for their unending love and support, and for encouraging and comforting me when things got tough. This thesis would not have been possible without their care and motivation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

As computations are increasingly offloaded to data centers, maintaining low latencies across servers is crucial. Particularly in distributed systems where communication is the backbone of all applications, minor increases in latency can add up to large costs in both money and energy.

Network latencies are incredibly important within data centers. As such, hardware improvements have led to drastic increases in data bandwidth over the past few years. However, other aspects of data center networking have not improved correspondingly. In particular, the Linux network stack tends to suffer from large user-kernel boundary overheads. Copying data from kernel space, where networking logic typically lies, to user space, where user applications run, often takes many cycles. As a result, large overheads are accumulated each time a process must cross the user-kernel boundary. Data copy from kernel to application buffers alone can take up the majority of CPU cycles in certain use cases[1]. To take full advantage of the latest advances in networking hardware technology, we must also maximize the bandwidth of our networking applications by minimizing these kernel overheads.

Current approaches to reducing datacenter latency are costly. One approach is to re-

place the POSIX (Portable Operating System Interface) API, which is a standardized set of functions that should be implemented within an operating system, with specialized APIs [2], [3]. While these specialized APIs provide much better performance than calling POSIX functions directly, developers must refactor their codebases to get such performance gains. Alternatively, other approaches require specialized hardware, as is the case with Remote Direct Memory Access (RDMA)[4]. Once again, although RDMA decreases latencies significantly, it requires organizations to purchase costly RDMA-enabled network interface controllers (NICs) and modify their applications to be compatible with RDMA. While previous research has successfully minimized data center latencies, it does so at a high upfront cost.

Rather than migrating the network stack logic out of the kernel and into the user space, we propose a new system, known as LakePlacid, for migrating the most commonly executed path in the application logic out of user space and into the kernel. By executing the most frequently called application path (or paths) within kernel space, we aim to achieve state of the art performance. Moreover, in providing differential treatment to the most commonly executed code paths, we implicitly prioritize different classes of traffic in a pattern similar to QoS in network paths. In utilizing an automatic compiler-based framework to reduce the user-kernel boundary overheads in networking applications, LakePlacid is able to see significant performance improvements with minimal developer effort.

As originally implemented, LakePlacid promises state of the art performance results with a lower upfront cost. However, it also suffers from subpar correctness and security arguments. While the security of LakePlacid is beyond the scope of this thesis, it is incredibly important to have strong guarantees that code augmented by LakePlacid maintains its original semantics. In particular, although LakePlacid handles most packets in the kernel, some packets fail during kernel processing and must be handled in user space instead. When this happens, the original LakePlacid restarted the processing of a given packet, potentially executing side effects a second time as a result, changing the semantics of the program. To remedy this, we need the ability to seamlessly migrate between execution in kernel space and user space

to preserve the program intent in all cases.

1.2 Contributions

LakePlacid is a compiler-based framework to achieve great performance boosts with little developer effort. While the specialized application succeeds in achieving microsecond-scale latency for its most frequent use case, if the execution diverges from its expected path at any point during runtime, the execution stack must be unwound so the fallback implementation may be called instead. Ideally, we would not need to revert the stack to an earlier state and queue up the request to be processed by the fallback implementation. Rather, we could determine how the current branch within the smaller, optimized kernel module — known as the EliteCode — matches with a branch in the unoptimized fallback implementation. Then, instead of entirely reverting the stack, we could update it to match what the fallback implementation would expect at this point in the code and continue execution from the middle of this implementation, not the beginning.

The ability to seamlessly migrate execution from kernel space to user space yields several key improvements. To begin, we expect to see marginal gains in performance, for code is no longer executed redundantly when an assumption about the types of expected packets no longer holds. Secondly, resuming execution from some saved state helps to ensure correctness by preventing extra side effects from occurring.

The primary benefit of seamlessly continuing execution in the fallback path from some saved execution state is that this provides stronger correctness arguments for LakePlacid. As a simple example of this, consider the case of a server that tracks metadata about the requests it receives before responding to them. Further, let us assume that this server's workload primarily consists of HTTP GET requests. In this case, the Elite code only contains the logic for handling certain GET requests, and all other requests will be handled by the fallback logic in user space. Assume this server has just received a POST request. In this case,

```
1  inline int lp_likely(int condition, ...) {
2      if (!condition) { abort(); }
3      return 1;
4  }
5
6  int handle_request(request_t req) {
7      increment_request_count(req);
8
9      if (lp_likely(req.http_method == HTTP_GET), ...) {
10         // Respond to the GET request
11         ...
12     }
13     /* else branch optimized out in Elite code */
14 }
15
```

Figure 1.1: A simple sample of a LakePlacid application which incurs additional side effects on the fallback path.

`abort()` will be called on line 9, triggering the request to be queued up to be resolved on the fallback path instead. In LakePlacid's original implementation, the fallback implementation would dequeue this request and begin processing from the top of `handle_request`, calling `increment_request_count` unnecessarily in the process. Executing side effects additional times like this goes against the original semantics of the program and leaves the application in an unintended state. While inaccurate metrics might not pose the largest concern, re-executing program logic like this could cause a whole host of problems, and potentially open up additional attack surfaces which are exploitable by adversaries. This thesis describes techniques that allow the request to be processed from the line of code at which it was aborted, avoiding the problem of re-executing program logic in the fallback path. Instead, with the changes described in this thesis, execution would resume from like `else` branch on line 13, following the expected program semantics.

Additionally, in comparison to previous approaches to on-stack replacement, the techniques described in this paper are quite generic and open up the ability to apply on-stack replacement in new use cases. As stated by the title, we contribute a method for implementing on-stack replacement across kernel and user space, which could create new opportunities for migrating pieces of other tasks into the kernel. Further, previous methods for on-stack

replacement are typically only applicable at function or loop bounds, or they otherwise restrict the space of locations where execution state can be swapped between code versions. We contribute a general framework for on-stack replacement which can migrate execution state at arbitrary conditionals.

However, there are limitations to this approach. Most notably, by removing the invariant that execution state will only ever be moved at specific loop or function bounds, we lose out on the ability to apply a number of optimizations which would have otherwise been exposed by LakePlacid. For instance, after profiling a given codebase on a sample workload, the corresponding generated kernel module is much smaller than the original application code. In theory, this means that we should be able to optimize it more aggressively. For instance, we could reallocate registers on this smaller codebase to reduce the number of spills. However, to ensure correctness, we must enforce the invariant that register state is equivalent between these two code versions, making this more aggressive register allocation infeasible. There are many opportunities to continue optimizing the methodology proposed in this paper, and these are further discussed in Section 5.1.

Chapter 2

Background

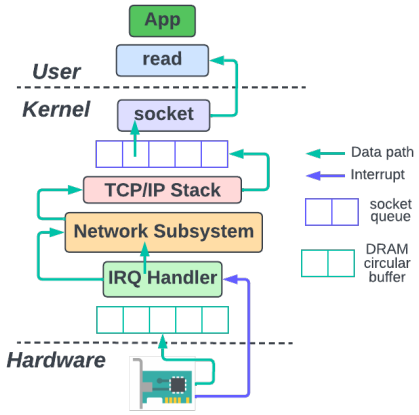

Figure 2.1: A traditional Linux TCP networking stack where data must be copied from kernel to user buffers before being processed by the end application.

2.1 User-Kernel Boundary Overheads in the Linux Networking Stack

A typical Linux networking stack consists of logic across three key components: the network interface card (NIC), the kernel networking stack, and the user level end application. In this setup, incoming packets are received by the network interface card (NIC), which then stores

the packet data in a receive ring buffer (RX). This ring buffer is memory-mapped to be accessible by the kernel. Processing of the network and transport protocol layers is handled the kernel, and the processed data is eventually made available to applications in user space via the POSIX networking API. User applications make syscalls to access incoming data and transmit outgoing responses, and all actual application logic occurs in user space.

In most cases, the separation of concerns between the hardware NIC, the kernel network stack, and the user application provides more benefits than drawbacks. The application developer is provided with a clean, portable POSIX socket interface that abstracts away much of the network complexity. Kernel networking stacks are time tested and well maintained, and any updates to the kernel network stack improve networking across the entire breadth of applications reliant upon it.

However, the layers of abstraction separating the user from the internals of the network stack also add their own overheads, which can become impermissible in high-performance environments. As more computing has shifted into large data centers, hardware vendors have noticed this trend and improved networking hardware accordingly. In fact, with the rate at which networking hardware is improving, IEEE is currently working on setting the standards for 1.6Tbps Ethernet[5], and major networking vendors currently offer hardware to support 400Gbps Ethernet [6], [7]. On the software side, in such distributed computing environments, applications like Memcached[8] often have service times of only a couple microseconds. Between the ever-increasing speeds of the network hardware and the end applications, the software network stack must operate at or below microsecond scale to avoid becoming a bottleneck.

With such high performance hardware and applications, warehouse scale distributed systems often find themselves paying many precious cycles to the dreaded "datacenter tax", which is comprised of low-level building blocks like memory allocation, data movement, and kernel operations. In fact, this "datacenter tax" can comprise nearly 30% of cycles[9]. Looking into this at more detail, Peter et. al. measured that, in a minimal UDP echo server

setup, 26.19% of packet processing overhead comes from context switches and data copies between the user and kernel[10]. And another study found that in an optimized kernel network stack with a single flow of data being transmitted between a sender and receiver, nearly half of CPU cycles were spent copying data between the user and the kernel[1]. In latency-sensitive environments, these user-kernel overheads are unacceptable, leading to a proliferation of strategies for overhead mitigation.

2.2 Kernel Bypass

High performance networking applications may opt to avoid the kernel altogether, in a technique aptly known as kernel bypass, in order to circumvent kernel overheads such as user-kernel copies and context switches. To remove the kernel from the data processing path, the network and transport protocol logic must be moved elsewhere. Typically, a more specialized network stack will be implemented in user space and spin poll for data directly from the NIC, whereas a traditional kernel network stack would receive interrupts from the network device.

Kernel bypass sees significant performance improvements from removing the overhead of syscalls and data movement; however, these may not be the only contributors. Kernel bypass systems are often more specialized than their intentionally generic kernel network stack counterparts, which allows for more streamlined data processing. Additionally, especially in such high throughput environments, interrupt requests are less efficient than spin polling[11], [12].

2.2.1 Software Solutions

Kernel bypass is a well-researched area with a wide breath of implementations. User-level network stacks like DPDK[13], OpenOnload[14], Linux Kernel Library[15], and F-Stack[16] provide user-level libraries for developers to use as building blocks when creating their own

specialized user network stack implementations.

DPDK in particular has been adopted in both industry and academia, acting as an important component in several other academic kernel bypass systems. For instance, Shenango[17] builds on top of DPDK by noticing that kernel bypass systems often waste CPU cycles spin polling. To solve this, it dedicates a privileged core to drive granular core allocation adjustments based on the status of the network. With this change, it is able to obtain the benefits of a user level networking stack with improved CPU efficiency.

Other solutions implement kernel bypass by rethinking the traditional operating system. Demikernel[3] is a new operating system architecture that enables engineers to easily utilize kernel-bypass in their datacenters by exposing a set of library operating systems (libOSes) that developers can swap in and out based on their application's needs. And, Arrakis[10] solves the issue of kernel overheads by removing the kernel from the data path, and instead providing isolation guarantees via hardware IO virtualization. The kernel is then reengineered to handle only infrequent operations on the control plane, thus ensuring standard kernel guarantees are still in place but with much higher throughput.

The world of kernel bypass techniques is extensive and growing, and it includes many more systems, like NetVM[18], netmap[19], and mTCP[20] just to name a few.

2.2.2 Hardware Solutions

While most kernel bypass systems operate by migrating core kernel functionality into user level libraries, another solution is to push more functionality onto the network hardware, thus allowing network devices and user space libraries to share the load of packet processing.

Remote Direct Memory Access (RDMA) is a technology which provides direct access between the main memory of two servers. Any networking protocol is implemented on the NICs themselves, allowing packet communication to bypass the kernel entirely to obtain higher throughputs and lower latencies [21]. Due to the incredible performance of RDMA, it has been used in high performance computing scenarios for years[22]–[24]. More recently,

smartNICs have also risen as a means of offloading packet processing from the CPU and increasing the programmability of the network fabric. These advanced network hardwares boast impressive speedups; however, they require physically installing specialized NICs in a datacenter, and they often are not compatible with the default Linux network stack, adding a learning curve for developers. Additionally, the design space of NIC-offloading possibilities is huge and requires careful consideration to obtain the greatest speedups[25].

2.3 Migrating Applications to the Kernel

While kernel bypass techniques have been well researched, and even adopted in industry, they come with a number of drawbacks. For one, they often require that at least one core is dedicated to spin polling for new packets, reducing overall CPU efficiency. Bypassing the kernel also bypasses its application isolation and security mechanisms, which could create new bugs or vulnerabilities.

Furthermore, even after the initial implementation, kernel bypass techniques require maintenance to stay competitive with the mainline Linux kernel. The Linux kernel is extremely well known and well maintained, with thousands of contributors[26]. Software-based kernel bypass systems in particular do not have nearly as much active support and development. And, because they implement at least portions of the network stack typically handled by the kernel, new optimizations added to the mainline Linux kernel must be manually re-implemented within these bespoke network stacks.

As such, another technique for mitigating user-kernel boundary overheads is to instead migrate user level packet processing code into the kernel. This is most frequently done via eBPF[27], a technology which allows programs to run in the kernel without losing out on security. In particular, the eXpress data path (XDP)[28] technology is a solution fully integrated into the Linux network stack which gives user-defined programs access to safely process packets in the kernel. These programs are verified with eBPF to maintain the security

of the kernel. By migrating logic into the network, end applications still get all the benefits of running on a well maintained kernel, and they don't have to dedicate a core to spin polling. Although the performance improvements are impressive, there is a learning curve involved in the development of eBPF programs, and in order to maintain the security guarantees it promises, there are some constraints on the kinds of programs that can be expressed.

2.4 LakePlacid

This work builds upon an existing system known as LakePlacid[29]. LakePlacid is a compiler-based framework designed to optimize datacenter applications using a couple key observations. First, the Linux network stack tends to suffer from large user-kernel boundary overheads which contribute significantly to overall datacenter latencies. LakePlacid removes these overhead costs by migrating networking programs to kernel space, as is illustrated in Figure 2.2. Moving part of the application code into the kernel improves performance in a few ways. First, by installing application code as a kernel module, packets can be handled completely within the kernel, avoiding the cost of copying data from kernel space to user space buffers. Additionally, processing packets in both the kernel and user space allows for better throughput. Packets with longer processing times are handled in parallel in user space, freeing up the kernel to continue processing other packets quickly.

LakePlacid relies on migrating networking applications into kernel space; however, networking applications are often large, complex systems which must be able to handle all kinds of requests. To reduce the amount of code moved into the kernel, LakePlacid relies on a second observation here. Depending on the use case, only a tiny fraction of a given code base may be responsible for the vast majority of traffic and functionality. So, to make networking applications more efficient, only the most trafficked lines of code need to be optimized. For instance, a content distribution network server may serve primarily static requests. Therefore, even though it has the ability to serve a wide variety of requests, only code associated with

static requests would need to be optimized. As another example, GET requests comprise over 99.8% of the workload for Facebook's most-requested Memcached pool (as of 2012)[30], so optimizing for GET requests alone could yield significant performance boosts. In three phases, LakePlacid determines which blocks of code are trafficked most frequently in normal operation, optimizes these common code blocks and packages them in a kernel module, and finally deploys the application, now specialized to handle its common use case much more efficiently. The kernel module is referred to as the *EliteCode*, and the user space code is referred to as the *Fallback*.

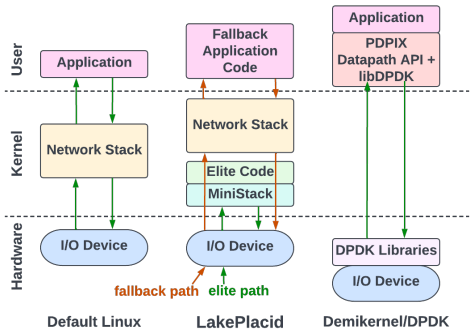

Figure 2.2: An example of the EliteCode and Fallback paths in LakePlacid. In comparison to other approaches which migrate networking logic out of the kernel, LakePlacid instead migrates application code into the kernel. Most packets are handled fully within the EliteCode (shown in green), so the kernel-user boundary rarely needs to be crossed.

2.5 On Stack Replacement and JIT Compilation

On Stack Replacement (OSR) is a well known technique within compilers, dating back to 1994 when it was first prototyped in SELF[31]. In this original implementation, on-stack replacement was used for deoptimizing code during debugging, as the optimized code may have undergone too many transformations for a developer to be able to understand how the optimized instructions correspond to the original logic. As recovering state from any point in the program was considered an impractically difficult problem to solve, SELF defined *interrupt points*, which were located at method prologues and at the end of loop bodies,

where an optimized program could be deoptimized for debugging.

While on-stack replacement was originally pioneered to simplify debugging, it is most often used as an optimization strategy in modern implementations. For instance, JIT compilers which must balance both compilation time and runtime can take advantage of OSR by quickly compiling an unoptimized version of the code to begin, and then recompiling optimized versions of only the most frequently executed methods as needed. Once the optimized version of some method has been compiled, information on the stack must be replaced to switch execution from the slow method to the new, fast version. On stack replacement can also be used to compile an optimized version of some code based on a set of assumptions, and then deoptimize as needed when these assumptions are validated. Today, many production VMs implement these techniques, such as V8[32], SpiderMonkey[33], and the Java HotSpot VM[34].

More recent publications have attempted to generalize OSR and make it more broadly applicable. For instance, D'Elia et al[35] proposes a framework for on-stack replacement at the compiler's intermediate representation level. Among other features, it supports transitions from optimized to deoptimized code, transitions at arbitrary points in the code, and transitions between provided code and code which was generated with profiling information known at the time of compilation. The work described in this thesis similarly supports all these kinds of transitions. Essertel et al[36] built on D'Elia's work by presenting a method for implementing it in source-to-source compilers. OSR has historically been used primarily in virtual machines, and thus were able to provide novelty by viewing it from a higher-level.

Although OSR is a well-known technique, my work differs from all implementations mentioned above in a key way. Specifically, my solution must handle both the switch between two different versions of code and the switch from kernel space to user space. This introduces a number of challenges. For one, the two code implementations reside in different address spaces, adding the need for address checks and translations. Additionally, we want to be able to dispatch packets to the Fallback path when necessary while continuing to process

new packets within the EliteCode, so the unoptimized version of the code must be able to queue up saved stack information to restore and execute. This is in comparison to standard techniques where the stack is immediately restored in some deoptimized code, without any necessary queuing.

Chapter 3

Implementation

3.1 Overview: System Workflow

LakePlacid relies on profiling data in order to make justified decisions about how to select code to be included in the Elite path. As such, LakePlacid compiles and executes source code in a few phases: the scouting phase, the code generation phase, and finally, the specialized execution phase.

3.1.1 Scouting Phase

My work as detailed in this thesis generally assumes the existence of known profiling data and did not require changes to the Scouting phase as originally implemented in LakePlacid. It is still included here, as it is helpful background to understand the entire system workflow. In the scouting phase, LakePlacid generates a *trace* for each incoming packet. Traces are files which log every time an if-else branch is executed, along with whether that branch evaluated to true or false. These traces are then used to determine which code paths should be included in the Elite code.

The scouting phase consists of a single source-to-source Clang tool which takes a preprocessed C file as input and outputs a copy with all of the if/else branches augmented to log

information whenever the branch is executed. Other kinds of control flow, like while loops and switch statements, are not modified. The augmented code is then run to profile the typical workload and generate a directory full of traces. These traces are organized into sets of equivalent execution paths and then greedily synthesized into a manifest which classifies each branch to be *likely, unlikely*, or *unknown*. In typical workloads, most packets follow the same or similar code paths. Profiling an application's networking workload enables us to make assumptions about which branches will be run and greatly reduce the amount of code moved into the kernel as a consequence.

3.1.2 Code Generation Phase

Once a manifest file has been generated to specify which code paths should be included in the Elite code, LakePlacid generates an executable to be run in user space, along with a set of object files to be linked and installed into the Linux kernel as a loadable kernel module. Each component of this process will be explained in greater detail in later parts of this thesis.

Generating the User Space Executable

The user space version of the generated code acts as a fallback, only being called when the Elite code cannot handle a given request. However, it is still heavily modified to ensure that the Elite and Fallback code paths are compatible with one another. Like the Elite code, source-to-source code transformation passes[1] are applied to the Fallback code to augment it with information from the generated manifest, add address checks and translations to indirect function calls, and modify global variables to be accessed from a table synchronized between the Elite and Fallback implementations. Code for storing and accessing both global variables and function addresses is then generated. The Fallback code is then linked to a

[1] Adding these code transformation passes to code built with arbitrary build systems added significant developer effort to the original version of LakePlacid, and as part of this line of work, we also previously developed tooling to automatically augment build systems with additional code passes or bash scripts. Modifications were made to that program as necessary throughout the development of this thesis, but more details on it can be found in prior work online[37].

number of helper C files which contain the code for maintaining the task queue ring buffer, handling leftover `lp_likely` calls, and so on. Finally, this code is called via a driver which coordinates the initialization of the Elite code kernel module and allocates memory to be shared between the Elite and Fallback code.

Generating the Kernel Module

The Elite code kernel module is generated in a very similar fashion to the Fallback code. It is also modified via three source-to-source code transformation passes and linked to many helper C files to handle stack allocation and queuing, memory management, and on stack replacement. However, because it is compiled as a kernel module, it is compiled with a number of restrictions which make it compatible with the kernel.

3.2 On Stack Replacement

Upon receipt of a packet, a new stack is allocated as described in Section 3.3, and the current thread of execution switches over this new stack. As an example, assume incoming packets are handled by a function `process_pkt`. Rather than calling it directly, LakePlacid dispatches `process_pkt` to be called within an assembly function, called `save_and_switch` which also handles saving the current state and swapping to a new stack. In executing each request on its own separate call stack, these call stacks can then be dispatched to finish executing in the Fallback implementation if needed. Sample code demonstrating this logic is provided in Figures 3.1 and 3.2, and much of the following explanation refers to variable and function names as defined in these code samples.

As shown in Figure 3.3, following standard x86 practices, `save_and_switch` first sets up a new frame on the original stack. It then pushes the current register values onto the original stack and saves the stack pointer in a variable (`elite_thread_rsp`) before setting the stack pointer to point at the top of our newly allocated stack (step 2). On the new stack, it saves

```c
/* Elite Code main.c */
void* elite_thread_rsp;
int process_pkt(struct process_pkt_params_t params);

int main() {
    ...
    for (;;) {
        listen_for_packets();

        stack_t new_stack = allocate_stack();
        int res = save_and_switch(params,
                                  process_pkt,
                                  new_stack,
                                  elite_thread_rsp);

        if (res == 0) {
            // This packet could be resolved in the Elite code
            free(new_stack);
        }
    }
    ...
}


```

Figure 3.1: An example of the main processing loop in the Elite code. Notice that rather than calling the packet processing logic directly, it is passed through another function in order to be executed on a separate stack.

```c
/* Fallback Code main.c */
void* fallback_rsp;
ring_buffer* stack_rb;

int main() {
    ...
    for (;;) {
        stack_t stack = poll_queue(stack_rb);
        int res = resume_execution(stack);
        if (res == 0) {
            free(stack);
        }
    }
    ...
}


```

Figure 3.2: An example of the main processing loop in the Fallback code, which waits for new call stacks to finish executing.

the previous stack pointer and finally calls `process_pkt` (step 3).

Notice that we have saved the original stack pointer in two locations; it exists both at the top of the new stack and as a variable. Being able to restore this pointer by just popping it off the top of the new stack once execution has completed is the cleaner and more efficient storage mechanism, and we prefer this option when possible. However, execution may diverge from the expected Elite path at any branch, and it is common for this divergence to happen from within a deeply nested function call. As such, trying to trace back up the stack to locate the old stack pointer in these cases would add unnecessary complexity, and it is worthwhile to store the old stack pointer in a potentially redundant variable for simplicity.

Once the Elite code has begun executing on an allocated stack, there are two possible paths that can be taken while processing a packet. Depending on whether or not the incoming packet can be fully handled by the Elite code, the packet will either be processed successfully in the kernel, or it will be queued up for execution in user space.

In the majority of cases, packet processing will be resolved solely by the Elite code. The Elite code will simply be executed on the new stack under normal x86 calling conventions until it eventually returns back to `save_and_switch`. The stack pointer is then reverted back to the kernel stack address by popping it off of the top of the allocated stack. All of the register values when `save_and_switch` was called have been saved on the kernel's execution stack, and the kernel state is restored by popping the register values off based on the order in which they were pushed onto the stack. Finally, `save_and_switch` returns `0` to indicate that the packet was processed fully within the Elite path, and as such, the allocated stack may be deallocated and used for a later call.

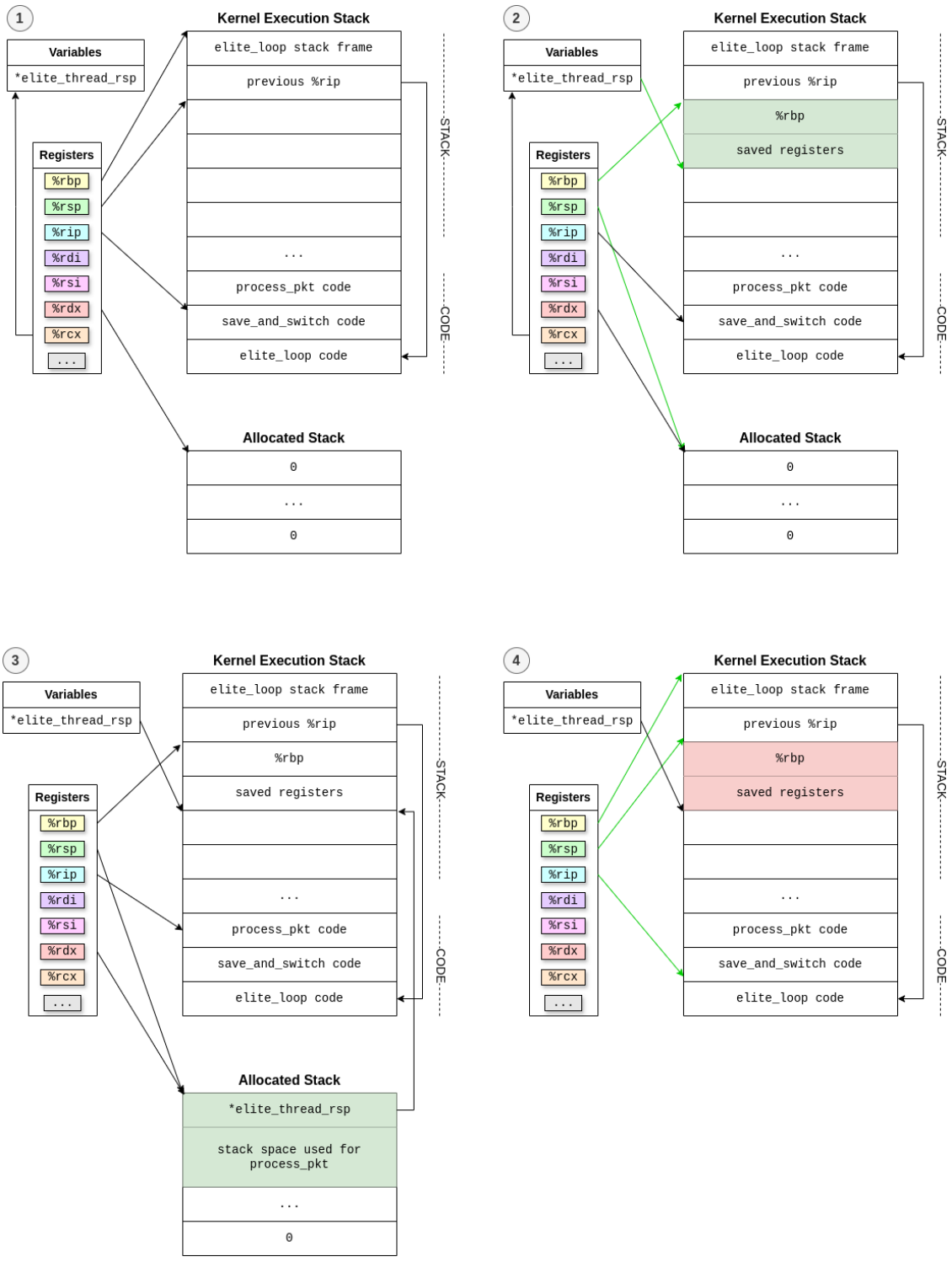

Figure 3.3: The different stages of the stack layout for a packet which can be fully handled by the Elite code. The packet handling logic is executed with a newly allocated call stack. Once complete, the stack pointer is reverted to the original kernel stack and the allocated stack is freed.

While the majority of packets should be processed in the straightforward method described above, things become more complex when a packet cannot be handled by the Elite path alone. Once again, a new stack will be allocated and `process_pkt` will be executed in this stack. However, at some point during execution, one of LakePlacid's assumptions on which code paths are usually taken will be shown to be incorrect, either via an `mpns_likely` branch being called with a `false` condition or an `mpns_unlikely` branch being called with a `true` condition. When this happens, LakePlacid must save the stack and queue it to be processed in user space.

In this case, as displayed in Figure 3.4, LakePlacid queues the current stack up to be executed by the fallback path in user space. To follow through this logic in more detail, LakePlacid first pushes the register values from the time at which it aborted execution onto the current stack (step 2). It then enqueues the current stack to be processed in user space eventually, and it restores the stack pointer to the correct location in the kernel, which was saved by `elite_thread_rsp` (step 3). Lastly, the old register values are restored, and `save_and_switch` returns `1` to indicate that the queued stack should not be deallocated.

When the saved stack is ready to be dequeued and executed in user space, a similar process is followed whereby the executing user space thread first saves its stack and registers (Figure 3.5, step 2) before setting the stack pointer to the enqueued stack and restoring the saved registers (steps 3 and 4). From here, execution can mostly continue normally. However, all the return addresses currently on the stack point to addresses in kernel space, which would trigger an exception. As such, all `ret` instructions called in user space are augmented with a translation step to translate them from a kernel instruction address to the corresponding user instruction address. This process is explained in much greater detail within section 3.4.2. Similarly, accesses to global variables are modified to ensure the kernel and user versions stay coordinated, and the user space memory allocator is overridden so values added to the heap in kernel space are accessible to the user space program. The mechanisms for both of these techniques are detailed in sections 3.5 and 3.6 respectively.

Figure 3.4: The stack layout when a packet cannot be handled by only the Elite code and must abort execution. In this case, the current state is saved on the allocated stack, and this stack is queued up for fallback execution. Then, the thread resumes listening for new packets with the original kernel call stack.

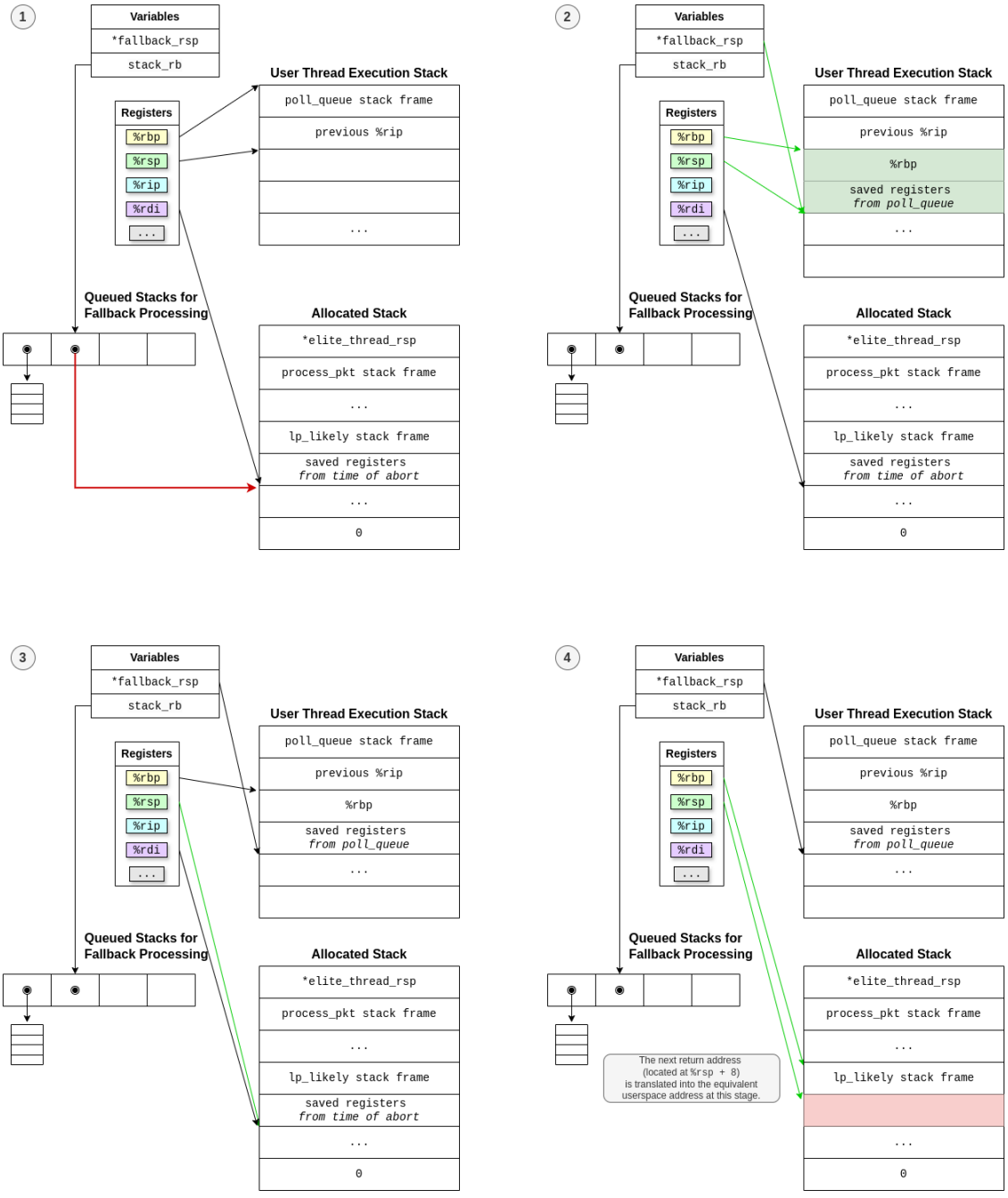

Figure 3.5: The different stages of the stack layout on the Fallback path. When a new stack is queued for execution, the polling fallback thread saves its own state and then switches to the new stack. From there, it restores this call stack's state and executes the relevant user space code.

Once the packet has been processed fully within user space, the thread of execution will eventually try to return to `save_and_switch`, as this is the outermost frame in the stack. However, the `save_and_switch` function described above tries to pop all the register values off the stack, which would result in unknown values filling up the registers here. So instead, the user space application is linked to a different version of `save_and_switch` which will instead restore the stack pointer from the polling user space function (which has been saved as a global `fallback_rsp` in user space) and then pop the register values from that stack instead. This sequence is illistrated in Figure 3.6. Finally, from the original user space stack, the polling function continues checking for newly enqueued stacks to process.

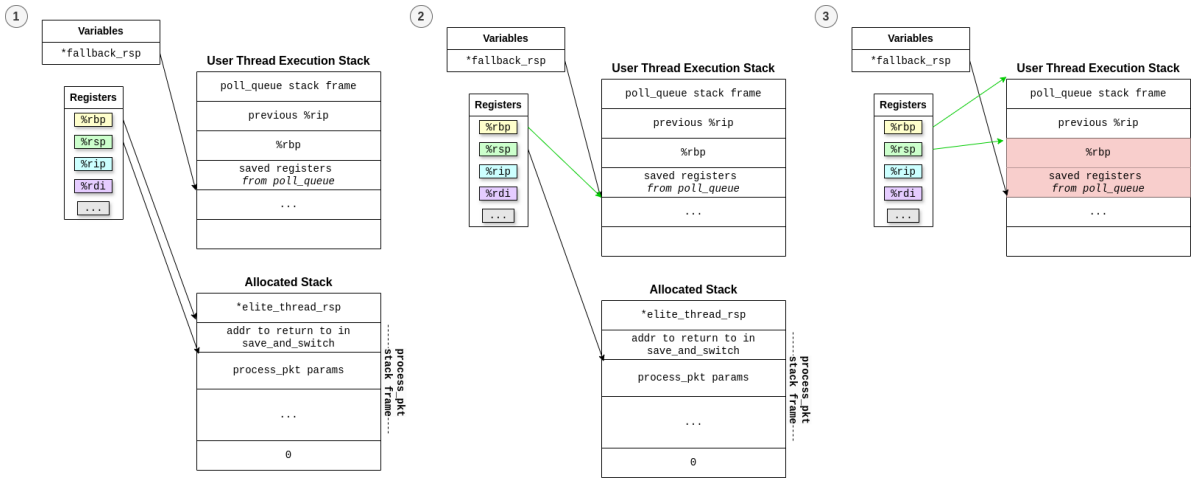

Figure 3.6: Once a queued call stack has finished executing, a linked userspace variant of `save_and_switch` restores the fallback process to its original call stack.

3.3 Stack Management and Queuing

Each time an incoming packet begins being processed, the thread of execution switches to a new stack. By executing each packet on a separate call stack, state can be easily saved and restored in the case that the Elite code cannot fully resolve the packet and must queue it to be completed in the Fallback implementation.

Because the each allocated call stack may need to be used by the user space fallback implementation, a large region of shared memory is first memory mapped to be accessible by both the kernel space Elite process and the user space Fallback process. We then implement a bitmap allocator which simply allocates fixed sized stacks and keeps track of which stacks are free with a bitmap. We are unable to accurately predict the stack size needed to process each individual packet, and we do not want to add the overhead associated with dynamically resizing a call stack. Therefore, we must statically assume the maximum stack size needed to process any given packet, and allocate this much memory to each newly created stack. Because of this fixed-size allocation constraint, the bitmap allocator is a reasonable choice over more sophisticated allocation strategies.

Call stacks that get transferred to the fallback process are inserted into a globally accessible ring buffer. The fallback process continuously polls this ring buffer to check for new stacks to dequeue and process.

3.4 Address Translation

3.4.1 Indirect Function Calls

During execution, there are two instances in which a function address may be incorrect. First, on the kernel side, indirect function calls may refer to the wrong address. C programs make heavy use of indirect function calls as a means of adding abstraction layers and reducing redundant code. Particularly in nginx, there is a common paradigm of setting function

pointer values during startup based on the configuration file. As the setup is only run once, it does not need to be included in the Elite path and is instead run in user space before the kernel driver is started. This means that most function pointers point to user space addresses, rather than kernel ones. To prevent the kernel implementation from unintentionally calling user space functions, each indirect call in the kernel implementation is wrapped in a checks to see if the called function pointer is in kernel or user space. If it is in user space, LakePlacid relies on the function tables described later in section 3.4.3 to translate the address into the corresponding kernel space address.

3.4.2 Return Address Translation

As was alluded to previously in section 3.2, the second time when a function address may be incorrect is when processing a packet on the Fallback path. In the specialized execution phase, every time a packet is received, a new stack is created and the kernel begins processing this packet. If this packet then violates an assumption from the Elite path and diverges in execution, the stack is queued for execution on the Fallback path. The stack contains much necessary state on the current execution path, including the return addresses of enclosing functions. However, these addresses refer to locations within the kernel's Elite binary, which are inaccessible by the Fallback binary.

To remedy this, every time the Fallback path encounters a `ret` assembly instruction, it attempts to translate this return address into the equivalent address within the userspace implementation before jumping to it. This is accomplished via an LLVM MIR pass which inserts an extra function call before each return to check the location of the return address, and update it to the proper address within user space if necessary. While the user space implementation is augmented with an extra function call, this call is unnecessary for the kernel path and would just add extraneous cycles. However, an additional `ret` instruction is still added to the kernel space implementation, so the instruction addresses line up between the user and kernel versions.

Maintaining alignment between instructions within corresponding user and kernel functions is currently necessary as we calculate equivalent instruction addresses based on their offset from the to top of the function. And, once again, this translation is possible due to the globally accessible function tables created at the start of runtime.

Address Translation Calculations

To simplify the address translation between the kernel and user space implementations, we enforce two key constraints.

Let f'_k be the kth instruction of function f in the Elite implementation, let f_k be the kth instruction of function f in the Fallback implementation, and let $\&i$ denote the address of some instruction i. With this notation, instruction addresses can be defined in terms of the start of the function in which they reside. ie. $\&f'_k = \&f'_0 + k * (\text{instruction size})$.

1. For every kernel instruction f'_k, there exists an equivalent user instruction f_k. The inverse is not necessarily true, as the set of functions in the Elite code is a strict subset of the set of functions in the Fallback code.

2. For any given kernel instruction, the state of the stack and all registers must exactly match the state of the stack and registers of the equivalent user space instruction, and vice versa.

These first constraint allows us to calculate the mapping from a kernel instruction f'_k to a user instruction f_k directly with $\&f_k = \&f_0 + (\&f'_k - \&f'_0)$. And, the second constraint allows us to continue execution from the equivalent user instruction without needing to make any adjustments to the state of the registers or stack.

These constraints add a significant amount of simplicity to our model, but they also inhibit our ability to implement certain performance optimizations, as is further discussed in section 5.1. Additionally, these constraints add somewhat awkward requirements to the

code, like how additional `ret` instructions are added to the Elite code assembly to make it align with the Fallback assembly.

3.4.3 Runtime Address Tables

The calculations for both indirect function call translation and return address translation rely on knowing the instruction address for the start of each function, in both the Elite and Fallback versions, at runtime. However, the Elite code and Fallback code exist as two distinct binaries and have no implicit knowledge of the other's location in memory. To remedy this, arrays of function addresses are filled in during setup and passed between the Elite code and Fallback via the `ioctl` system call. These addresses are become accessible via automatically generated functions of the form `function_name__addr()`.

3.5 Global Patching

Unlike functions, which may have differences in implementation between the user space and kernel space versions, global variables should be shared between the user and kernel. In applications like nginx, global variables are often only changed during setup and remain static for the rest of execution. Sharing globals between the two implementations saves a minor amount of memory, and more importantly, it ensures that values set by the user space implementation – especially during setup – are accessible by the kernel module. All global variables are allocated in user space. Like function addresses, an array of global variable addresses is populated during setup and communicated to the Elite code via an `ioctl` system call.

Accessing and potentially modifying globals in two parallel threads immediately raises concerns that this will introduce bugs from the ordering of reads and writes. This concern is especially relevant given that kernel threads may pause execution and enqueue their state to be completed later at any arbitrary if-statement. However, production networking applica-

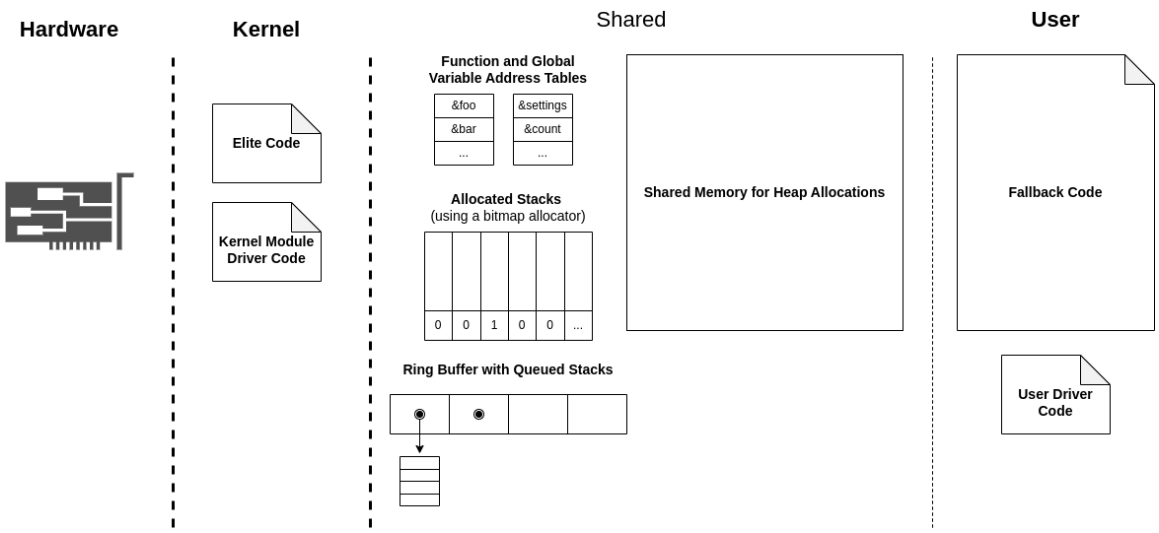

Figure 3.7: Overview of key LakePlacid components in memory. Notice that much of the data allocated in user space is shared for use by the Elite code in the kernel.

tions are nearly always designed to be multithreaded for performance reasons. Any bugs that could arise from accessing globals from multiple threads would have already been considered and mitigated by the authors of the original source code.

3.6 Memory Allocation

Switching between the Elite code and the equivalent Fallback code is only correct if all the relevant state is maintained during this stick. Global variables are synchronized between the two code implementations and all local stack-allocated variables and register are available via the transferred stack. However, this is missing on last piece of state: heap-allocated values. To remedy this, during setup, the user space implementation memory maps a large section of memory to be used by both the kernel and user implementations, and their memory allocators are overwritten with a simple bump allocator that uses this shared region of memory.

3.7 Driver Design

All of the components described above are coordinated via drivers in both user and kernel space. Referring to 3.7, the user space driver first allocates memory for the call stacks, ring buffer, function and global variable address tables, and the shared heap memory allocator. It then initializes the kernel driver via an `ioctl` syscall and shares information about all the allocated memory with the kernel. From there, the kernel driver starts listening for packets using the logic in the Elite code, and the user driver creates a new thread to poll the ring buffer for new stacks to process.

Chapter 4

Evaluation

While the overarching goal of LakePlacid is to achieve state of the art networking performance with less developer effort, the goal for this thesis is to add lightweight support for on-stack replacement to LakePlacid, enabling seamless transitions between the Elite code and Fallback code. Previous benchmarking for LakePlacid used additional optimizations that either were cut from this proof of concept due to time constraints, like the integration of LakePlacid's MiniStack network stack, or because they violate the invariant that Elite instructions should be in alignment with equivalent Fallback instructions (sec. 3.4.2). As such, until these kinds of optimizations can be integrated in with the system changes proposed here, we do not expect to see performance improvements compared to the original LakePlacid work. Rather, we want to verify that the changes made here do not add significant overhead to the Elite code path.

With additional optimizations enabled to help this meet the original LakePlacid performance, we would then still expect some additional overhead from switching the stack before and after executing each incoming request on the Elite path. As the time taken to allocate a new stack, switch to it, and then revert back to the original stack after execution is incurred every time the Elite code is able to complete the execution of some incoming request, keeping these overhead costs low is of utmost importance. In particular, with LakePlacid configured

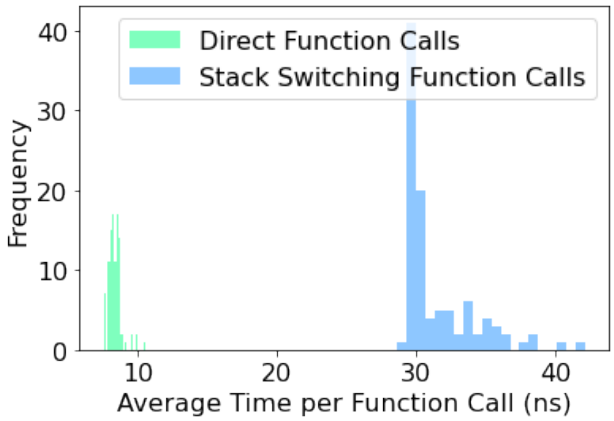

Figure 4.1: The added overheads of switching stacks before and after function execution. It is clear that the additional stack switching mechanism added some overhead, but the additional measured overhead is only approximately 23ns.

correctly, we anticipate that a large majority of incoming requests would be solely handled by the Elite code, so maximizing performance there is a key concern.

The following experiment was performed on an Ubuntu 20.04 virtual machine with a Linux 5.15 kernel. The host machine operates on a 6 core Intel i7-8750H Processor running at 2.2GHz, and the virtual machine has been allocated 2 vCPUs and 4GB RAM.

To quantify the overheads associated with dispatching the execution of the Elite code onto a separate stack, I first timed how long it takes to call a minimal function from within a loadable kernel module 10,000 times. The function being called is so small that a singular function call timings were dominated by the `rdtsc()` time, and calling the same function many times in a loop is a more accurate scenario to emulate listening for new requests, too. I then performed this experiment again, this time calling the minimal function through our custom dispatching mechanism, which also handles acquiring a stack, switching onto it, and then freeing it post-execution. I repeated this test 100 times to obtain the numbers shown in Table 4.1 and Figure 4.1.

Based on Table 4.1, the added overhead of dispatching the Elite code onto a separate stack is only about 23 nanoseconds. The bitmap allocation strategy helps keep this overhead low, as allocation only requires iterating over a bitmap until a 0 is found and using that index to

Statistic	Direct Function Calls	Stack Switching Function Calls
Mean	8.34328	31.51113
Standard Deviation	0.4902	2.6907
Median	8.28535	30.28385
Min	7.5351	28.6494
Max	10.5667	42.1686

Table 4.1: Average Time per Function Call (ns)

identify the stack's location. Note that this overhead would be substantially higher if we ran out of free stacks to use. Additionally, we currently don't zero out stacks before using them, which greatly improves performance, but also increases the attack surface and could pose security issues. Overall, as our goal is to stay under the microsecond scale, 23 nanoseconds is a permissible amount of overhead.

Chapter 5

Future Work

To properly scope this project, we constrained this problem and left out further ideas that would be out of reach given the time restriction, leaving open a number of possible future research directions for LakePlacid.

5.1 Optimizations

As explained in section 3.4.2, the on-stack replacement technique described here relies on functions to align perfectly in terms of instruction selection and stack state between the user and kernel versions. This strict constraining made translating kernel instruction addresses to user space instruction addresses much simpler, as the translation could be expressed just based on an instruction's offset from the top of the function. However, many classical compiler optimizations were also limited or completely disabled as a consequence.

Currently, because `lp_likely` and `lp_unlikely` are defined as external assembly functions which are not incorporated until link time, the compiler can't reason about their specifics. This leaves open potentially the most obvious performance improvement: the `else` branch of `lp_likely` if statements (and conversely, the `then` branch of `lp_unlikely` if statements) can be optimized out. This would require some extra minor bookkeeping to still be able to translate between kernel and user instruction addresses properly, but in return, it

would reduce the size of the kernel module, potentially improving instruction cache locality in certain use cases. And, combining this optimization with additional compiler passes, like dead code elimination, could further streamline the Elite code. That said, applying further compiler passes to this more pruned version of the Elite code would require extensive bookkeeping to be able to translate the instruction addresses correctly and reconstruct the state of the stack and registers as needed.

5.2 Security Concerns

For the purposes of this thesis, we assume LakePlacid would primarily be used the speed up communication between servers within secure datacenters, which would make security exploits less likely. However, they are still a matter of grave concern, and a number of implementation decisions in LakePlacid increase the attack surface for an adversary. In particular, the user application itself might be buggy, which potentially creates serious weak points when portions of that buggy code is migrated into the kernel. Additionally, our implementation relies on lots of communication between the user and kernel. The kernel even occasionally dereferences user addresses directly, as is the case with global variables. Future work could consider taking advantage of technologies like eBPF[27] rather than loadable kernel modules to improve the safety of our techniques, as well as adding additional checks for malicious activity on the fallback side.

5.3 Dynamic Profiling

Currently, source code is only profiled by LakePlacid during the initial setup. As such, if the most common use case changes over time, LakePlacid cannot adapt to fit these new needs. However, the ability to instrument build systems automatically opens up a new set of possibilities to solve this issue. For instance, we can modify LakePlacid to continue profiling code during the specialized execution phase, and using an instrumented version

of the build system, we can automatically recompile EliteCode as needed throughout a networking system's lifetime.

In order to update the EliteCode dynamically based on changes in request patterns, we must be able to continue gathering profiling information during the specialized execution phase. We could use existing instrumentation to continue profiling the EliteCode while it runs; however, the profiling code adds too much overhead to be viable in a production context. In constrast, because the Fallback path already incurs a higher latency and should not be executed frequently, it is worthwhile to profile the Fallback code.

In implementing a system which dynamically updates the EliteCode as needed, we must strike a careful balance between computing new manifests relatively infrequently in order to keep overhead low, while also calculating these often enough that we are alerted immediately when the request patterns begin to change. Future work will detail how to find this balance and update the EliteCode at the correct frequency.

Chapter 6

Conclusion

As network hardware improves, high performance networking applications are increasingly bottlenecked by kernel overheads. To solve this, LakePlacid compiles a smaller, optimized version of networking applications to be run in the kernel – thus mitigating overhead costs when copying data between the kernel and user space – while leaving the original application in user space as a fallback. Taking advantage of domain knowledge, LakePlacid achieves performance gains on par with state-of-the-art approaches on several benchmarks while utilizing source-to-source compiler passes to reduce its upfront cost. However, it also suffers from correctness issues when a packet fails in the Elite path and must be executed in the Fallback path instead, for restarting the processing of a packet causes side effects to be re-executed unnecessarily. This thesis proposes and implements a set of changes to LakePlacid which utilize on-stack replacement (OSR) to properly migrate execution between the kernel Elite code and user Fallback code. Being able to seamlessly transition from a kernel space instruction to its equivalent user space instruction improves performance and guarantees correctness by eliminating redundant computations and side effects. The solution proposed allows deoptimization to occur at any branch. Moreover, it efficiently handles the transition between kernel and user space with low overheads which make it feasible for use even in extremely latency-sensitive environments. More work is needed to integrate the method proposed here

with all of the optimizations included in the original implementation of LakePlacid, but the work described in this thesis solves a major problem for LakePlacid, making it more feasible in real-world scenarios.

References

[1] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 65–77, ISBN: 9781450383837. DOI: 10.1145/3452296.3472888. [Online]. Available: https://doi.org/10.1145/3452296.3472888.

[2] A. Kalia, M. Kaminsky, and D. G. Andersen, "Datacenter rpcs can be general and fast," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19, Boston, MA, USA: USENIX Association, 2019, pp. 1–16, ISBN: 9781931971492.

[3] I. Zhang, A. Raybuck, P. Patel, *et al.*, "The demikernel datapath os architecture for microsecond-scale datacenter systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21, Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 195–211, ISBN: 9781450387095. DOI: 10.1145/3477132.3483569. [Online]. Available: https://doi.org/10.1145/3477132.3483569.

[4] *A rdma protocol specification*, Oct. 2002. [Online]. Available: http://www.rdmaconsortium.org/.

[5] IEEE. "Ieee p802.3dj 200 gb/s, 400 gb/s, 800 gb/s, and 1.6 tb/s ethernet task force." (2024), [Online]. Available: https://www.ieee802.org/3/dj/ (visited on 08/14/2024).

[6] Cisco. "Cisco nexus 400g - the next frontier for cloud networking." (2020), [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/data-center/high-capacity-400g-data-center-networking/brochure-c02-741700.html (visited on 08/14/2024).

[7] J. Networks. "400g & 800g: Network scalability that just keeps going." (), [Online]. Available: https://www.juniper.net/us/en/solutions/400g-and-800g.html (visited on 08/14/2024).

[8] dormando. "Memcached: Performance." (2016), [Online]. Available: https://github.com/memcached/memcached/wiki/Performance (visited on 08/14/2024).

[9] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, Portland, Oregon: Association for Computing Machinery, 2015, pp. 158–169, ISBN: 9781450334020. DOI: 10.1145/2749469.2750392. [Online]. Available: https://doi.org/10.1145/2749469.2750392.

[10] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The Operating System Is the Control Plane," en, *ACM Transactions on Computer Systems*, vol. 33, no. 4, pp. 1–30, Jan. 2016, ISSN: 0734-2071, 1557-7333. DOI: 10.1145/2812806. [Online]. Available: https://dl.acm.org/doi/10.1145/2812806 (visited on 06/18/2024).

[11] P. Cai and M. Karsten, "Kernel vs. User-Level Networking: Don't Throw Out the Stack with the Interrupts," en, *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 7, no. 3, pp. 1–25, Dec. 2023, ISSN: 2476-1249. DOI: 10.1145/3626780. [Online]. Available: https://dl.acm.org/doi/10.1145/3626780 (visited on 06/18/2024).

[12] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," en, *Communications of the ACM*, vol. 60, no. 4, pp. 48–54, Mar. 2017, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3015146. [Online]. Available: https://dl.acm.org/doi/10.1145/3015146 (visited on 06/18/2024).

[13] D. Project. "Dpdk." (), [Online]. Available: https://www.dpdk.org/ (visited on 08/14/2024).

[14] I. Advanced Micro Devices. "Openonload high performance user-level network stack." (), [Online]. Available: https://github.com/Xilinx-CNS/onload (visited on 08/14/2024).

[15] L. K. L. Project. "Linux kernel library." (), [Online]. Available: https://github.com/lkl/linux (visited on 08/14/2024).

[16] T. Cloud. "F-stack." (), [Online]. Available: https://www.f-stack.org/ (visited on 08/14/2024).

[17] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, pp. 361–378, ISBN: 978-1-931971-49-2. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[18] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms," en, *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, Mar. 2015, ISSN: 1932-4537. DOI: 10.1109/TNSM.2015.2401568. [Online]. Available: http://ieeexplore.ieee.org/document/7036139/ (visited on 06/18/2024).

[19] L. Rizzo, "Netmap: A novel framework for fast packet {i/o}," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA: USENIX

Association, Jun. 2012, pp. 101–112. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[20] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems," en,

[21] I. Red Hat. "Chapter 1. understanding infiniband and rdma." (), [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/configuring_infiniband_and_rdma_networks/understanding-infiniband-and-rdma_configuring-infiniband-and-rdma-networks (visited on 08/14/2024).

[22] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 295–306, Aug. 2014, ISSN: 0146-4833. DOI: 10.1145/2740070.2626299. [Online]. Available: https://doi.org/10.1145/2740070.2626299.

[23] Y. Gao, Q. Li, L. Tang, *et al.*, "When cloud storage meets RDMA," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, USENIX Association, Apr. 2021, pp. 519–533, ISBN: 978-1-939133-21-2. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/gao.

[24] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 202–215, ISBN: 9781450341936. DOI: 10.1145/2934872.2934908. [Online]. Available: https://doi.org/10.1145/2934872.2934908.

[25] P. M. Phothilimthana, S. Peter, M. Liu, R. Bodik, A. Kaufmann, and T. Anderson, "Floem: A Programming System for NIC-Accelerated Network Applications," en,

[26] Github. "Linux." (2024), [Online]. Available: https://github.com/torvalds/linux (visited on 08/14/2024).

[27] eBPF.io Authors. "What is ebpf?" (2024), [Online]. Available: https://ebpf.io/what-is-ebpf/ (visited on 08/14/2024).

[28] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66, ISBN: 9781450360807. DOI: 10.1145/3281411.3281443. [Online]. Available: https://doi.org/10.1145/3281411.3281443.

[29] A. Brahmakshatriya, M. Ghobadi, and S. Amarasinghe, "Lakeplacid: A compiler-based framework for transforming datacenter applications to the microsecond latency regime," unpublished, 2022.

[30] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12, London, England, UK: Association for Computing Machinery, 2012, pp. 53–64, ISBN: 9781450310970. DOI: 10.1145/2254756.2254766. [Online]. Available: https://doi.org/10.1145/2254756.2254766.

[31] U. Hölzle and D. Ungar, "A third-generation self implementation: Reconciling responsiveness with performance," *SIGPLAN Not.*, vol. 29, no. 10, pp. 229–243, Oct. 1994, ISSN: 0362-1340. DOI: 10.1145/191081.191116. [Online]. Available: https://doi.org/10.1145/191081.191116.

[32] What is V8? [Online]. Available: https://v8.dev/.

[33] SpiderMonkey. [Online]. Available: https://spidermonkey.dev/.

[34] M. Paleczny, C. Vick, and C. Click, "The java HotSpot™ server compiler," in *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, Monterey, CA: USENIX Association, Apr. 2001. [Online]. Available: https://www.usenix.org/conference/jvm-01/java-hotspot%7B%5Ctexttrademark%7D-server-compiler.

[35] D. C. D'Elia and C. Demetrescu, "Flexible on-stack replacement in llvm," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16, Barcelona, Spain: Association for Computing Machinery, 2016, pp. 250–260, ISBN: 9781450337786. DOI: 10.1145/2854038.2854061. [Online]. Available: https://doi.org/10.1145/2854038.2854061.

[36] G. M. Essertel, R. Y. Tahboub, and T. Rompf, "On-stack replacement for program generators and source-to-source compilers," in *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2021, Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 156–169, ISBN: 9781450391122. DOI: 10.1145/3486609.3487207. [Online]. Available: https://doi.org/10.1145/3486609.3487207.

[37] K. Mohr, "Transforming datacenter applications to microsecond latency with profile-guided optimization." [Online]. Available: https://www.mit.edu/~kmohr/assets/lakeplacid.pdf.